Automatic analysis of ancient languages

Darius Blasband
darius@phidani.be

First release : September 23, 1996
Version : 1.06
Last modification : January 2, 2000
Printing Date : January 2, 2000

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is amazing what people have in mind when they use the word “language”. If they’ve had even loose contacts with the computer world, they’ll often understand “programming language”. Other people will refer to spoken languages, those alien things we call “natural languages” as if they appeared spontaneously out of chemical elements one commonly finds in the nature. Silicium, perhaps.

Languages can be found in far more instances in the computing world than just in programming. We have specification languages. We have page description languages [1]. We have HyperText Markup Languages (better known as HTML [2]). The sets of escape sequences recognized by printers and terminals are languages. With the recent advent of XML [3] [4], we even accept the idea that besides describing code – that is, executable things – languages can be used to describe data. Together with the move towards object-orientation, mobile agents, reflexive systems, the fronteer between code and data which used to be cut and clear is getting fuzzier by the day.

Basically, any software is about languages, or at least, a language. Turing machines – which are one of the most primitive views on computing – deal with languages. Np-completeness [5] is about languages. When dealing with a piece of software, the sequence in which operations can be performed define a language. To some extend, it is this degree of freedom of modern graphical user interfaces in terms of language – any operation can be invoked at virtually anytime – that makes them hard to develop, hard to maintain and hard to test.

When one reads dates from a database, these dates must conform to a format, to a sequence of signs, to a language. The Year 2000 problem can be seen as a language transformation issue. Addresses are defined by languages. Amounts are described by languages. Even names are described by languages. BIBTEX [6] [7] describes how the first and last name, as well as the initials can be put together to form an author’s name using an ad hoc language.

Somehow, after Jesus “Love is everywhere”, Freud “Sex is everywhere”, Rotschild “Money is everywhere” and Einstein “Everything is relative”, after Adele Goldberg or Bertrand Meyer “Objects are everywhere”, I’d be tempted to add: “Languages are everywhere” [8].

Languages can be made of textual elements, character or words. The set of characters can vary from language to language (APL [9], is an extreme example of this). Language can also be made of immaterial items (mouse clicks, external events). Languages can even be made of graphical elements. Some modern development tool vendors claim that they can deliver applications "without programming". This is of course nonsense. Programming is not about touch typing. It has nothing to do with the fact that a keyboard rather than a mouse is used as interface. Programming is about manipulating and combining abstractions, and the rest is irrelevant ("*l'intendance suivra*" would have added Napoléon Bonaparte). A few years ago, there was even an ad for a CASE tool, that showed pages of listing, with a big motto: "How much code did you write today ? How much good did it do ?". There was of course something of a tone to suggest that one should not write code any longer, but use their super-duper brand new revolutionary piece of software, to draw rectangles, clouds and arrows rather than to write text. I'd be tempted to add: "How many drawings did you produce today ? How much good did it do ?".

Fortunately, languages most software deal with are often simple enough not to require explicit language recognizers. Such applications can be written ad hoc, and the fact that they can be seen as language processors is merely an intellectual nice-to-have, a way of waving a hammer, claiming that all problems suddenly look like nails.

Then, some applications deal with languages that are too large and too complex to be developed without a deep understanding of their linguistic nature. It is true that most of these applications are – or at least, were – classical language translators, compilers and interpreters. Somehow, the person who associated "language" with "programming language" was not entirely wrong. In the beginning, there were semi-formal descriptions of languages, and recognizers for these languages were developed ad hoc. Some languages were reasonably easy to recognize, but due to the nature of most of the languages one had to deal with at the time, due to additional complexities such as error recovery or the lack of recursion to implement recognizers, also given the processing power of available hardware, it often was a daunting task.

Research went into the automatic production of recognizers, or at least, systematic ways of implementing hand-crafted recognizers. Efficient and generic algorithms were designed for well-defined classes of languages. Alternatively, we might also say that classes of languages were defined for which such algorithms could be produced. These algorithms were generic enough to support the automatic generation of the parser by a tool. The most famous result of this effort is of course `yacc` .

From there on, things should have been simple. It would have been sufficient to design new languages within one of these well-defined classes, to be able not to worry about writing recognizers any longer.

As it often seems to be the case, reality has been disappointing with regard to this charming even if unrealistic view on the world. This is due to several factors:

- These nice theories appeared late enough for older languages to have gained momentum, and odd things such as `COBOL` and `PL/1` are still in widespread use today. It is not always enough to have solutions for the problems that will arise tomorrow, even if relectant, one must also cope with the past.

- We now deal with other languages than programming languages only, and the classification theory we have at hand does not apply very well. We also sometimes deal with *variations* on existing languages. We must analyze *scaffolded* languages, for instance, when analyzing embedded SQL in `C` source code, or `C` source code *before* it has gone through the preprocessor, when one must recognize a mix of plain `C` constructs and preprocessor directives.

- Some languages such as Rexx [10] even take strong positions to claim that languages should be designed for the user's – the programmer's – convenience, not to ease the work of the implementor. Similarly, Python [11] [12] gives a semantic meaning to the indentation level, which comes in strong contradiction with the commonly accepted idea, supported by the parser generation world, that white space should serve no other purpose than presentation[1]. On the other end of the spectrum, Oberon [13]'s definition explicitely states that making the language LL(1) was a deliberate design decision aimed at making the life of the implementors easier. In practice, it seems hard to sustain that accepting such restrictions limit the language's expressivity in any noticeable way.

- `C` and `C++` have appeared long after the wide acceptance of language classification[2]. As this document will show, these languages are ill suited to automatic parser generation. They require extensive twiddling and user hooks. More on this later...

- The parser generators – mainly `yacc` – were designed with performance of the resulting parser in mind, at the expense of virtually any other factor such as convenience, modularity, quality, etc. For a long time though, hand-crafted parsers could deliver performance which was better enough to justify not using parsers generators. With the ever growing processing power of available hardware, this concern should disappear, even if Hanson and Fraser [14] still advocate hand-crafted parsers.

The purpose of this work is to propose a technique to produce analyzers for a wide class of languages, including languages that were not designed with parsing in mind, that is, for languages that might not match the grammar classes for which we do have parser generators ready.

This work is also complete in the sense that it is not limited to an academic study of theoretical interest. It includes a complete and industrial implementation of the parser generation tools. This description goes as far as providing the entire source code of the implementation in literate programming form [15]. Just as Tanenbaum noticed in [16], providing the source code measures the gap between theory and reality. Some theoretical issues, on which hundreds of pages of intricate articles have been written, account to less than a page in the final code. On the other hand, a working implementation must solve hundreds of sometimes amazing problems that are not even mentioned in the literature.

[1] While it is perhaps a bit early in the course of this paper to get in the body of gory details, it is worth noticing that Python is context-sensitive by the way it manages white spaces, while most parser generators are restricted to context-free languages only.

[2] Amazingly enough, these languages have been developed at Bell Labs, where `yacc` was developed as well, but it seems that some valuable information never went from one team to the other

1.1 Previous work

The idea of producing parsers for unusual languages did not occur to me as an isolated intellectual matter. This work is, in many respects, the follow-up of numerous previous works and ideas.

RYPP [17][3] was a project related to the development of the syntactical and semantical analysis of PL/1 [18] source code. The parser was hand-written in plain YAFL[4], and the extra level of abstraction provided by YAFL was supposed to allow Jean-Christophe Real to overcome the complexity of the language.

Within the limits of such a research project, RYPP was successfull. The goals that were defined when starting the project have been reached and exceeded. On the other hand, this project also led to some very valuable lessons:

- It is unreasonable to parse a complex language such as PL/1 manually, up to a point where one can wonder how the original PL/1 compilers might have been written, given the power of available hardware and the state of the art in compiler design at the time. While RYPP worked reasonably well, it was based on limiting assumptions about the way the language was actually used in real-world programs. Even so, given the dramatic level of ambiguity of the language, parsing PL/1 remains a significant undertaking.

- It is also unreasonable to structure a complex parser or compiler according to grammar rules which structure is dictated by some obscure – in the sense that it cannot be understood without a complete understanding of the parser generator internals – limitation such as LALR or LL. Compilers are complex pieces of code. Increasing their entropy by significant factors because of the parsing technology one uses only means one thing: the parsing technology is inadequate [19].

- There is still some work to be done at the lexical level. Language processing theory is definitely not fashionable anymore. Even more so, its restriction to regular languages is considered as a problem of the past. However, when one wishes to apply this theory to pathetic real-world examples such as PL/1, it simply does not work properly.

This work is also vastly inspired by cool[20] by Lionel Ferette. cool delivered a reasonably functional YAFL-based compiler-compiler. Somehow, this thesis is nothing more than the production development that follows the cool prototype. We evaluated the shortcomings of that prototype, considered ways of extending it in terms of its versatility as well as of the class of languages it can handle. On the other hand, we only reuse cool's concepts, and the lessons that have been learned. The tool itself is entirely redesigned, and quite logically, so is its implementation.

The third tribute I am entitled to make is more anecdotic. Literate programming [15] (which is a better formalized version of [21]) is a technique developed by Knuth to produce code and the attached documentation in a single, integrated document. The

[3]RYPP stands for Reengineer Your PL/1 Programs

[4]What is meant by *plain* YAFL, as opposed to a supposedly not-so-plain form will be explained extensively in the course of this work.

examples he proposed were so impressive, that a few weeks after having read his book on the subject, we started the YAFL/Web project, that aimed at providing a YAFL-aware literate programming environment [22]. We finished the YAFL/Web project with mixed feelings. The tool that had been developed worked reasonably well, but writing literate programs soon proved to be quite difficult in practice. Reading Knuth's brilliant literate programs proved misleading: he made it seem so easy ! Literate programming requires more than just programming skills: it also requires literate and educational skills, that are far beyond the average programmer's ability.

So, when I decided to include the source code of the tools in this document, I knew that producing understandable source code was going to be a challenge of its own. From that point of view, my prediction has been right: literate programming is fascinating and rewarding. It is also extremely difficult, and frustrating when a piece of code that has been arranged quite thoroughly in order to be understandable is rejected by a reader as "plain nonsense".

The last tribute is to `YSTORM`[23]. In theory, a global optimization system as implemented by `YSTORM` is only an industrial production tool, that improves the performance of a system using global analysis. It should have no impact on design or even on the way things are implemented. In practice, it is an entirely different matter.

When one knows the optimizations that will be performed one can write much simpler code, much easier to understand, to maintain and to comment. In practice, the global optimization system prevented me from optimizing some tight loops myself, from using intermediate variable or from accessing attributes rather than using access methods. I am deeply convinced that global optimization improves the overall quality of the code I deliver in terms of structure and readability.

1.2 A word about bibliography

Any book written in the past 15 years, and that relates, even remotely, to compilers without referring somehow to [24], better known as the "Dragon Book", has been written on another planet. The Dragon Book is such an excellent reference that very few other books even try to come close. This paper is no exception. It also relies heavily on stuff that is described in the Dragon Book. Citing it occurs so often, that I thought it would make the paper more enjoyable if I replaced this reference by a pictorial representation: . Some of the techniques described in this paper (more specifically, in the implementation of the tools) are not of paramount innovating value, and have been used for years. In some cases, rather than finding back an original paper that no one will ever search back, I'll cite  to indicate that more information can be found in the Dragon book.

1.3 A word about engineering

Engineering is about compromises. Engineering is about dealing with the real world, with real machines, real data, and real time. Engineering is about making choices. Painful choices, sometimes. Depending on the final aim, different design decisions must be made for the sake of one issue and at the cost of another. This universal (and depressing) law of computer science applies even more to language processing systems. When dealing with highly ambiguous languages for which the inputs to parse are limited in size – the knowledgeable reader will recognize natural language processing in these characteristics – an algorithm that guarantees cubic worst-case performance together with a very unfavourable constant factor is acceptable. When dealing with inputs of tens of thousands of words, especially if it is only *locally* ambiguous, using such a heavyweight system is totally out of the question.

By providing the full implementation, by aiming at using the tools in production systems, this paper is also an exercize of engineering that goes beyond the limits of a theoretical study. In some places, ideas that would have been applicable in theory have been abandonned because they were hardly implementable, or because their benefits were not worth the increased amount of work they would have required. For instance, an operation such as normalization of the grammar is strictly unreasonable if one plans to produce working parsers for real-world languages. Orders of magnitude must be taken into account. Resource consumption and performance must be predictable.

This emphasis on pragmatism is not gratuitous: this entire project started when I considered developing a new tool, called `COOL` (Not to be confused with `cool`). `COOL` is a universal `COBOL` parser, that should accept - understand: parse - any `COBOL` program, all dialects, all extensions. Then, `COOL` should be able to produce valuable reports about that program: portability, structure, complexity, what should be changed to port that program to this or that dialect, etc ... In the meantime, `COOL` has turned into a more generic and even more ambitious product named `RainCode`, described in page 920.

We can thus characterize the situation as follows:

- The input can be very large, up to 100000 lines.
- The grammar can be ambiguous, but these ambiguities are limited in scope. When in an ambiguous situation, the parser will usually be able to raise the ambiguity within at most a few hundred tokens.
- We are not aiming at defining yet another language class with a matching parsing algorithm, hoping that new languages will fit our model. The tools we design must be able to deal with existing code, no matter how ugly it can be.
- The grammars we will be dealing with tend to be large. In [25], Grosch mentions 1500 grammar rules for `COBOL`, even if such figures depend heavily on what one refers to as a *grammar rule*. Expressivity is not just syntactical sugar around a primitive formalism. It is of paramount importance. Delivering a formalism which can in theory cope with all the problems at hand is of little use if it complexifies the grammar specification by an order of magnitude. Besides providing the mere theoretical functionality, the tools must be usable as well.

1.4 A word about formal specification

This paper includes a thorough description of a complex mechanism, similar in concepts to the LR automaton mechanic described in (Don't you think such pictograms are a nice way of using references ?), but significantly more complex. Since some of the work is related to the actual behaviour of this mechanism, I thought it would be a nice idea to use the most formal description possible in order to avoid ambiguities. It is also true that in most of the projects I deal with in my professional daily life, I seldom have the opportunity to experiment advanced techniques such as highly mathematical notations, and I thought this thesis was a nice opportunity to try them out on a non trivial example.

I first considered using plain Z [26]. However, Z is quite awkward when it comes to the specification of OO concepts such as inheritance and polymorphism. Hence, I soon fell back to one of Z's OO extensions, namely ObjectZ [27] where OO concepts can be represented conveniently. By using ObjectZ rather than plain Z, my specification can be easily connected to the corresponding YAFL implementation. The entire compiler literature praises the virtues of reflexivity, that is, the ability for a system to describe itself. One of the results of this work was a working ObjectZ parser. It was nice to use this parser to parse the formal specifications that described the parser generator used to generate this ObjectZ parser in the first place.

1.5 A word about implementation

It is very common nowadays to consider implementation as an unimportant part of any software development process. A lot of effort is invested on improving analysis and design, and quite rightly so. It is true that in the past, we have been coding quite a lot, and thinking very little. On the other hand, neglecting implementation by considering it as a mechanical task with little if any added value is a bad mistake. Ultimately, the purpose of any design is to be executed, not just to exist as a design. Claiming that implementation has no impact on quality is just as a car which is built using chewing gum to glue pieces together, that will be a good car if its blueprints are good looking.

Implementation is also very rewarding. When writing code, we deal with some form of concrete reality, and it is quite fascinating to see how some parts of a design that might have looked complex at first sight, can be implemented in a straightforward way, while some more mundane part might requires lots of tricks and skills. Implementation is fun. Implementation is just like Lego [28]: "Daddy ! Look at the car I just built !".

Addressing implementation also raises industrial issues, such as real performance[5], or resource consumption. For instance, the entire literature about automatic parser generation makes extensive use of sets as basic data structure ([29], [30] and , among others). When one addresses implementations, one must also think of reasonable ways to implement such sets, and there is nothing mundane or trivial about this.

[5] As opposed to mathematic performance, which only compares orders of magnitude, neglecting constant factors that might have more importance in practice than these orders of magnitude.

I guess I must at least make a comment about YAFL [31]. Together with the trend toward more and more design, and less and less implementation, programming languages have gone out of fashion. It is pretty common to hear that the language is unimportant, and that a good design can be implemented in virtually any programming language. This is perhaps one of the reasons why object-orientation as a concept is quite widely accepted, while OO languages are still far from replacing more conventional systems.

I believe that such an approach is plain wrong. Depending on the language, its constructs, their semantics, one gets different levels of abstractions, one deals with entirely different concepts. The langage has a great impact on readability, on complexity, on maintainability. The language is the place where the coder looses control, and where an automated system (tuple $hardware \times operating\ system \times (compiler \cup interpreter)$) gets in charge. Depending on the language, some operations that are simple and efficient can become hard and unmanageable. For instance, the availability of a garbage collector or support for genericity can make a huge difference when it comes to the implementation of a working solution. The language still matters a lot.

The implementation presented as part of this paper is entirely written in YAFL [32] [31]. Of course, I designed the language in the first place, so I quite logically prefer using it whenever possible. On the other hand, I also believe that YAFL's high level of abstraction makes it a very good tool to develop complex data structures and algorithms such as those described in this paper. Here again, beloved reader, you ought to make an opinion of your own.

1.6 A word about literate programming

Providing large amounts of source as part of a thesis is a challenge, specially if one expects the reader to understand and even to enjoy reading it. So, instead of pushing it away in some appendix nobody would ever try to read, I included it in the text. I tried to use the best way of making this source code readable and understandable, by using literate programming, as supported by `noweb` [33] [34][6]. You will have plenty of opportunities to see what literate programming looks like in the course of this document. Let's just start by saying that it is made of some mix of code and the attached documentation in a single document, so that a well-structured documentation can be produced (as is included in this paper), and the source code can be extracted, compiled, and executed.

Literate programming also allows one to mix code and properly typeset math, so that one can easily see the connection between the underlying theory and the corrresponding implementation. This is even more the case when a formal specification language with complex typesetting such as ObjectZ is used. I believe it is of little use to describe something using a complex mathematical formalism, if one cannot provide the reader with the corresponding source, in an attempt to convince him (or her) that the code actually implements what has been described.

[6] I did not use the tool developed as part of the YAFL/Web [22] project, because it was aimed at *troff* [35] rather than LaTeX for typesetting, and because its implementation never left the prototyping phase, especially not to handle a huge and intricate document such as this one.

I realize that the amount of source code included in this work might discourage a potential reader from even trying to understand it, even in a superficial way. In order to emphasize the important things, and distinguish them from the rest of the code, I used pictorial devices. When a hand is shown in the margin it means that the source code contains essential information worth looking at. Hopefully, in such cases, there should be lots of textual comments as well, since these are the places where it all happens.

For those of the readers who would enjoy a roadmap with the highlights of this documents, the following table summarizes all the places in this thesis where this pictograms have been used.

When a bug is shown in the margin, the code is debugging code, that is, code that makes sure that some essential property of the system is fulfilled, or that outputs data structure onto a stream for debugging purposes. Amazingly enough, in [14], Hanson and Fraser remove *assertions* from the code before publishing it while I personally believe that such code tells more about a system than most informal textual descriptions.

When publishing code as I do here, one takes more care of issues such as elegance and quality. I am not just concerned for the mere executability of what I write, I also want it to be appealing to a reader who might not share my father-like adoration for my literary production. However, in some cases, implementation issues such as performance or another external constraint forced me to write things in a less than desirable way. Such cases, hopefully rare, are marked by a specific icon in the margin as well. Their purpose is to ensure the reader that these pieces of code are known to be poor, that they have not been left as a result of some kind of quick fix.

YAFL

Some pieces of code are strictly YAFL-specific. They make use of some special feature of the language, the standard library or even the YAFL compiler. They are also marked with an ad hoc pictogram, as shown here in the margin.

This work is not a theoretical publication with an example of a possible implementation. It is a documented production system. As such, it evolves, to support additional functionalities, to provide improved performance, or to increase robustness. Even if I tried to deliver a consistent whole, some parts are "under construction", marked with an easy to recognize glyph in the margin. Such parts are not completed, and sometimes, it is not even clear whether they will eventually be completed some day.

Last and least, when code is so boring and uninteresting that only its mother (understand, me) can find it even remotely attractive, yet another pictogram is shown in the margin.

`noweb` is language neutral. It provides a versatile literate programming environment that applies equally to any language. This generic approach naturally has drawbacks, namely, its lack of specialized YAFL-support. Some day where I simply could not go on working without yawning, I developed a quick and dirty YAFL beautifier for `noweb`.

This beautifier uses a different font for YAFL reserved words, and replaces some of them by more concise notations. For instance, a piece of code such as:

```
METHOD Max(p: ARRAY OF INTEGER): INTEGER;
  POST
    FOR_ALL x IN p :- RESULT >= x,
    THERE_IS x IN p :- RESULT = x;
```

is typeset as:

METHOD `Max`$(p$: ARRAY OF INTEGER): INTEGER;
  POST
    $\forall\ x \in p \bullet$ RESULT $\geq x$,
    $\exists\ x \in p \bullet$ RESULT $= x$;

This beautifier[7] also adds an additional level of abstraction for the reader. In YAFL, accessing the i^{th} element of an array `Arr` is written as `Arr[i]` while accessing the

[7] I do not resist the temptation to say that this beautifier relies extensively on `lexyt`, which is one of the tools described in this document, thereby moving the reflexive nature of this work even further.

i^{th} element of a container class – list, bitmap – `Cont` is written as `Cont.Get(i)`. Since there is little true difference between arrays and container classes in terms of their semantics, the beautifier will replace `Arr[i]` by $\texttt{Arr}_i$, and `Cont.Get(i)` by $\texttt{Cont}_i$, abstracting the difference for the reader entirely. The difference remains of course at the `noweb` source level in order to be able to produce a compilable YAFL source code, but the reader who is just interested in understanding how the tools work is not bothered with such low-level detail.

1.7 A word about me

Any technical writer knows about the awful dilemna of dealing with oneself. "The author", "We" or a systematic use of passive voice are possible ways of avoiding writing "I" or "me". For instance, in [36], Klaas Sikkel goes through circumvoluted formulas to explain that "we" is merely a literary way of say "I".

Whenever applicable, I prefer using I[8]. I use "we" explicitely when talking about work, reflections, and ideas that have come from my colleagues and I, together as a group. When referring to my own work, I try to avoid the plural form. There are basically two reasons for this:

- Writing "I" is shorter than writing "we". On a few pages, no one would feel the difference, but on such a large piece, saving one letter on every "I" rather than "we" accounts to quite some time. Of course, I could have also used a LaTeX macro to just type `\I` and see "we" or even "the author" printed. I am not sure whether this would have saved me any significant amount of work, because the "\" key is placed on an awkward position on azerty keyboards.

- I believe there is something special about this document. It is not a report about work that has been done. It is not a post hoc rationalization. It actually *is* the work itself. What you read, beloved reader, is the material I've been working with on an almost daily basis for almost three years. You'll be reading something very intimate. I'm not giving a few formulas, together with a vague claim about a working implementation. It's all here. The work as displayed here can be criticized extensively since it contains the most intricate details. I believe it would be a major *faute de goût* to maintain an artificial distance with a piece of work which is in fact so intimate.

All in all, beloved reader, enough of this chit chat. Here we go.

Welcome in my world.

[8] You would not have believed me anyway if I wrote "I prefer using we" [37]

Chapter 2

Lexical models

In the beginning, there was nothing.

Then, we got an alphabet.

$[\,Alphabet\,]$

The exact nature of this $Alphabet$ set is of little theoretical importance. It can be ASCII characters, binary digits, EBCDIC, or even some form of pictorial representation, for instance, when dealing with a graphical representation of purpose-specific code using an ad hoc low-level CASE tool.

Most if not all theoretical introductions to language theory are based on the existence of an abstract or concrete alphabet. The definition of finite state automata (for regular languages) and context-free grammars (for context-free languages) that appear in Hopcroft and Ullman [29][1] make an explicit reference to the alphabet concept.

It is concrete when the input codes represent something almost physical, at the collating sequence level. When saying that an identifier is made of an alphabetical character followed by a sequence of zero or more alphanumerical characters, we deal with a concrete alphabet, referring to the physical representation of these characters on the machine. On the other hand, it is abstract when the codes of the alphabet are just codes with no direct relationship with any external representation. For instance, claiming that a block starts with a keyword `BEGIN` and ends with a keyword `END`, we assume that there is some representation of these symbols in our system, but the exact nature of this representation is of little importance.

The purpose of lexical analysis as formalized here is to transform this stream of elements of $Alphabet$ into a stream of elements of a higher level of abstraction. In some cases, this new stream, in turn, can be used as input and serve as $Alphabet$ for another layer, yielding a scaffolding of lexical analyzers.

The exact nature of the input stream is not essential. It can be a file, a memory-based data structure, it can be any flow of information described by $Alphabet$ through some form of communication channel.

[1] For the earliest references, [29] cites McCulloch [38] for regular languages and Chomsky [39] for context-free languages

$$\begin{array}{|l}
Input : \mathrm{seq}\ Alphabet \\
InputLength : \mathbb{N} \\
\hline
InputLength = \# Input
\end{array}$$

[2]So, this is the input.

Let's now describe a token. A token is a contiguous piece of the input, a slice, attached to a code, defined as an element of $TokenCode$. This code will identify the abstract property of the token, independantly from its position and length.

$$[\ TokenCode\ ]$$

Usually, this set is represented using some form of discrete type.

In order to simplify the various algorithms and specifications, two special token codes EOF and $ERROR$ will prove useful:

$$\begin{array}{|l}
EOF, ERROR : TokenCode
\end{array}$$

We can now describe a token as referring to a slice of the input and holding a $TokenCode$[3]:

Token

$$\begin{array}{|l}
Code : TokenCode \\
Position : \mathbb{N} \\
Length : \mathbb{N} \\
\hline
Length = 0 \Leftrightarrow Code = EOF \\
Position + Length \leq InputLength + 1 \\
Position = InputLength + 1 \Leftrightarrow Code = EOF
\end{array}$$

The definition of $Image$ makes use of $Slice$ (See page 978).

$$\begin{array}{|l}
Image : \mathrm{seq}\ Alphabet \\
\hline
Position \leq InputLength \Leftrightarrow Image = Slice(Input, Position, Length) \\
Code = EOF \Leftrightarrow Image = \langle\,\rangle
\end{array}$$

$Cover$ returns the set of the indices in the input which are covered by the token.

$$\begin{array}{|l}
Cover : Interval\ \mathbb{N} \text{(See page 979)} \\
\hline
Cover = MakeInterval(Position, Position + Length - 1) \text{ (See page 979)} \\
Code \neq EOF \Rightarrow Cover \subset \mathrm{dom}\ Input
\end{array}$$

[2]The seq Z operator denotes a sequence, formalized as a partial function from a contiguous interval of integers starting at 1 onto a target set (in this example, $Alphabet$).

[3]I slightly extended the plain ObjectZ formalism, by allowing my classes to contain more than one state subschema (holding attribute definitions and their attached invariants) to make large classes with numerous attributes and comments more readable.

At this stage, we put no restriction on $Token$s relative to each other: they can overlap, they can have the same position, the same code, etc. On the other hand, we'll soon need the ability to compare $Token$s.

Tokens can be compared, using $TotalOrder$ (See page 977).

$$\_ \leq \_ : Token \leftrightarrow Token$$

$$(\_ \leq \_) \in TotalOrder$$
$$\forall\, a, b \in Token : a.Position < b.Position \Rightarrow a < b$$

Following intuition can lead to bizarre results: at first sight, one might be tempted to claim that a shorter $Token$ is smaller (understand: occurs before in a sorted list) than a longer one. In this context, when comparing two lexemes at the same position, we must take the *longest* first, reproducing `lex`'s [40] behaviour.

$$\forall\, a, b \in Token : a.Position = b.Position \land a.Length > b.Length \Rightarrow a < b$$
$$\forall\, a, b \in Token : a.Position = b.Position \land a.Length = b.Length \land a.Code < b.Code \Rightarrow a < b$$

Now that we can compare $Token$s with each other, we can instanciate the $CompareSequences$ schema (See page 978) to compare sequences of $Token$s lexicographically.

$CompareSequences[Token]$

We can now define a generic lexical analyzer as:

$Lexer$

$iseq$ as used here for $CurrentTokens$ is a unique sequence, defined by:
$s \in iseq\, K \Rightarrow \forall\, i \neq j \in \mathrm{dom}\, s \bullet s(i) \neq s(j)$
It is a sorted sequence of all $Tokens$, describing all the possible interpretations of the input at a given position. Of course, when dealing with common lexers such as those produced by `lex` [40] (See $Lexer_0$ (See page 21)), $CurrentTokens$ will be a sequence of at most a single $Token$.

$\mathit{WhiteSpace}$ is a set of atomic intervals that can be safely ignored, representing typically white space and comments. $\mathit{Skippable}$ is defined as a set of intervals that can be skipped, as they are the union of disjoint and contiguous intervals of $\mathit{WhiteSpace}$.

$$
\begin{array}{l}
\mathit{CurrentTokens} : \mathit{iseq}\ \mathit{Token} \\
\mathit{Skippable}, \mathit{WhiteSpace} : \mathbb{P}\ \mathit{Interval}\ \mathbb{N} \\
\hline
\mathit{CurrentTokens} \in \mathit{Sorted}[\mathit{Token}]
\end{array}
$$

(See page 978)

$$
\begin{array}{l}
\forall\, w \in \mathit{WhiteSpace} \bullet \\
\quad \forall\, p \in w \bullet p \leq \mathit{InputLength} \\
\forall\, s \in \mathit{Skippable} \bullet \\
\quad \exists\, ws : \mathbb{P}\ \mathit{Interval}\ \mathbb{N} \bullet \\
\qquad ws \subset \mathit{WhiteSpace} \wedge s = \bigcup_{x \in ws} x \wedge \\
\qquad\quad \forall\, w_1, w_2 \in ws \mid w_1 \neq w_2 \bullet \\
\qquad\qquad w_1 \cap w_2 = \phi
\end{array}
$$

$\mathit{ReadCurrentToken}$ as defined here, is an operation. The convention in ObjectZ is to use a postfix $?$ for input parameters a postfix ! for output parameters, and a postfix prime (as in $\mathit{CurrentTokens}'$) to describe the state of an attribute *after* the execution of the operation while the plain attribute name without the prime refers to the state of the attribute *before* the execution of the operation.

ReadCurrentToken

$$\Delta(CurrentTokens, TokPtr)$$
$$Pos? : \mathbb{N}$$

$$Pos? \leq InputLength$$

The following constraint states that the tokens read at the current position must be at this current position, or can be reached starting at this current position after having skipped a number of characters considered as skippable (Typically, white space or comments).

$$\begin{aligned}
&\forall\, t \in CurrentTokens' \bullet \\
&\quad t.Position = Pos? \lor \\
&\qquad (t.Position > Pos? \land \\
&\qquad\qquad MakeInterval(Pos?, t.Position - 1) \in Skippable)
\end{aligned}$$

(See page 979)

Reading all the possible tokens at a given position resets the pointer to the current interpretation ($TokPtr$).
The new value of $Carnation$ is implied by its definition and the relationship it holds with $TokPtr$ (see below).

$$TokPtr' = 1$$
$$Carnation' = 1$$

$$CurrentToken : Token$$
$$TokPtr : \mathbb{N}$$

$$\begin{aligned}
&CurrentTokens = \langle\,\rangle \Rightarrow \\
&\qquad CurrentToken.Length = 0 \land CurrentToken.Code = EOF \\
&CurrentTokens \neq \langle\,\rangle \land TokPtr \notin \operatorname{dom} CurrentTokens \Rightarrow \\
&\qquad CurrentToken.Length = 1 \land CurrentToken.Code = ERROR \\
&CurrentTokens \neq \langle\,\rangle \land TokPtr \in \operatorname{dom} CurrentTokens \Rightarrow \\
&\qquad CurrentToken = CurrentTokens(TokPtr)
\end{aligned}$$

$Carnation$, as described here, is redundant as far as this specification is concerned, but it is the artefact used by the implementation, which does not build the entire $CurrentTokens$ sequence on every position on which it stops, but rather, a two dimensional table, indexed by token length, and by *carnation*, which is the current interpretation number among the possible ways of understanding the current lexeme at the current position and with the current length.
Hence, $Carnation$ is the index of $CurrentToken$ among the subslice of $CurrentTokens$ with its current length.

For instance, let's consider a sequence of $CurrentTokens$ such as:
$A_5 B_5 C_5 D_4 E_4 F_3 G_3 H_2$
where the indices give the token length. When $TokPtr$ is 3, yielding C_5 as the current token, $Carnation$ is 3, since C_5 is also the third element of the slice of $CurrenTokens$ obtained by taking tokens of length 5 only.
When $TokPtr$ is 5, yielding E_4, $Carnation$ is 2, since E_4 is the second element in $CurrentTokens$ with length 4.

$Carnation : \mathbb{N}$

$Temp : iseq\ Token$ (See page 14)
ran $Temp = \{x \in \text{ran}\ CurrentTokens \mid x.Length = CurrentToken.Length\}$
$Temp \in Sorted(iseq\ Token)$

By construction, we have:

$CurrentToken \in \text{ran}\ Temp$

and $Carnation$ is defined by:

$Temp(Carnation) = CurrentToken$

$Advance$

$\Delta(CurPos, CurrentTokens, CurrentToken, TokPtr, Carnation)$

$CurrentToken \neq EOF$
$ReadCurrentToken \mid Pos? = CurrentToken.Position + CurrentToken.Length$

$BackTrack$

$\Delta(TokPtr, CurrentToken, Carnation)$

$TokPtr \in \text{dom}\ CurrentTokens$
$TokPtr' = TokPtr + 1$
$TokPtr' \notin \text{dom}\ CurrentTokens \Rightarrow CurrentToken'.Code = ERROR$

ReSync allows the lexer to restore a previous state of the lexer, typically, when undoing a *Shift* operation in the *Parser* class (See page 240).

ReSync

$\Delta(CurPos, CurrentTokens, CurrentToken, TokPtr, Carnation)$
$Pos?, Len?, Carnation? : \mathbb{N}$

The following expansion of *ReadCurrentToken* – equivalent to a method invocation in programming terms – does not take any explicit parameter, since no renaming is required: *Pos*? as received by *ReSync* is transferred as is to the expanded operation.

ReadCurrentToken

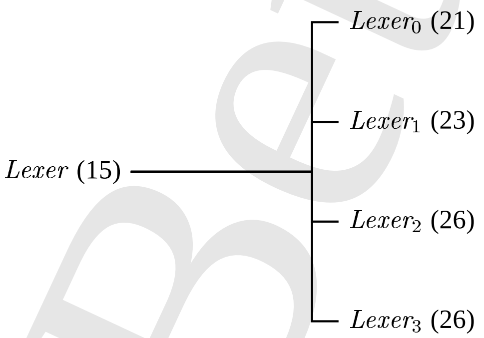
$$TokPtr' \in \operatorname{dom} CurrentTokens' \bullet$$
$$CurrentToken' = CurrentTokens'(TokPtr') \land$$
$$CurrentToken'.Length = Len? \land$$
$$CurrentToken'.Position = Pos? \land$$
$$Carnation' = Carnation?$$

INIT

When initializing a *Lexer*, it builds the list of tokens that can be understood at the beginning of the input.
It is by the way a little bit unfortunate that ObjectZ sequences start with index 1, while YAFL arrays and collection classes start with index 0.

$ReadCurrentToken \mid Pos? = 1$

Figure 2.1 Inheritance tree for the various classes of lexers



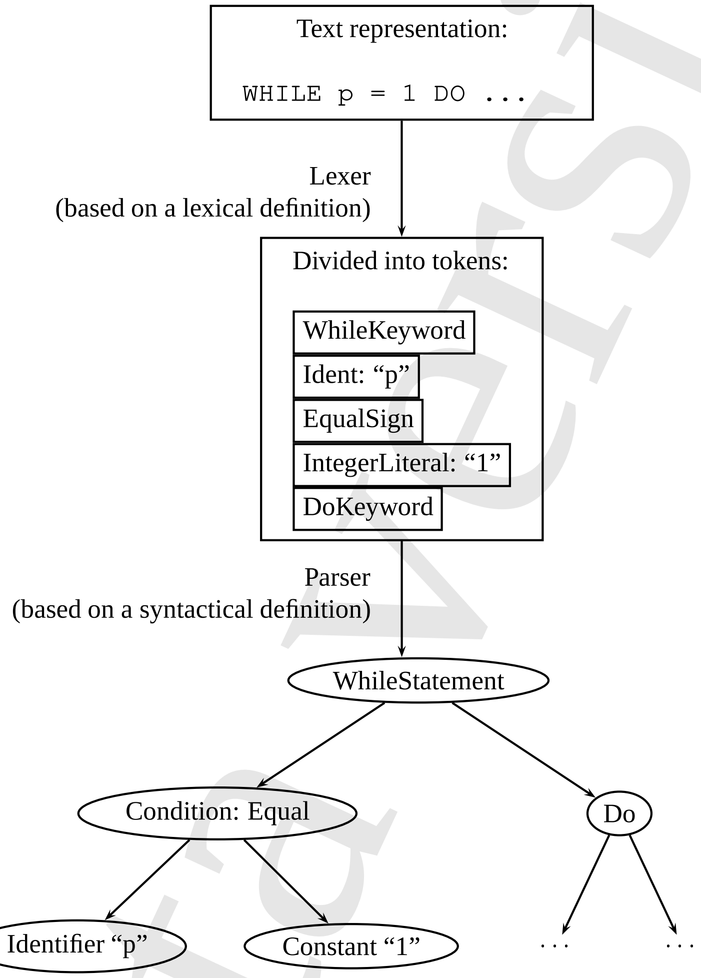
Based on this definition where we have factored out the properties that will be common to other kinds of lexical analyzers, we can now define lexers that match the *classical model*.

In this model, the input only supports a single interpretation, a single division into tokens. In this context, a parser can be seen as a machine that, given a context-free language definition and a input generates a tree that satisfies two conditions:

- Its structure matches the language definition. The parse tree must be a possible production of the language grammar rules
- Its fronteer (in left to right, infix order) must match the input list made of tokens of the language's alphabet.

This tree can only be generated if the input is valid, since the existence of this tree is the criterion for input validity.

Figure 2.2 The classical model for the division of labour between lexical and syntaxic analysis



In any case, the division of labour between lexical and syntactical analysis is based on the presumption that lexical analysis provides a sequence – a list – of tokens. These tokens are part of the syntactical analisys’s alphabet. The very idea of an alphabet implies that this list can be produced without referring to the upper-layer parser that will take this list, and produce the corresponding tree. In figure 2.2, the information flow is unidirectional from top to bottom.

$Lexer_0$

$Lexer$

This is exactly `lex`'s behaviour: at each position, there is at most a single token possible.

$\#CurrentTokens \in \{0, 1\}$

In practice, it might prove useful to consider scaffoldings of more than two layers. For instance, in `COOL`, when dealing with large numbers of `COBOL` dialects with different lexical conventions and different approaches to line-handling, it might prove useful to first normalize the input to a common representation before passing it to the lexical analyzer. However, the main issue remains the same: this entire theory and technology is based on the idea of the *separation* of these passes.

2.1 Shortcomings

It soon appeared that languages that were designed before the golden days of language theory were hard to parse and did not match the model described in [29]. This sections lists a few of these languages together with the associated problems one encounters when attempting to parse them. Some of the languages listed here could be expected by the experienced reader as raising interesting issues, while some others are more of a surprise.

2.1.1 `PL/1`

`PL/1` has always had the reputation of being a very complex and poorly designed language. This opinion applies equally to syntactical issues as well as to semantic issues such as pathetic type coercion mechanisms.

First, just as `FORTRAN`, `PL/1` does not support reserved words, and the example shown on figure 2.6 applies to `PL/1` as well. On the other hand, `PL/1` has problems of its own.

Figure 2.3 `PL/1` lexical problems

```
DECLARE (I,K,L,M,N,O,P)
```

Depending on the token that follows the closing parenthese, the statement listed in figure 2.3 can be a declaration of several variables with implicit type `BIN FIXED 15`, or a call to a function called `DECLARE`, with severals arguments provided.

Since there is no limit to the number of variables that can be declared within a `DECLARE` statement, nor to the number of parameters that can be given to a function, distinguinshing between these two statements requires a potentially infinite lookahead.

The example in figure 2.3 shows that `PL/1` cannot be parsed using a parser with finite lookahead. In other words, some form of backtracking is required. However, is one introduces some usage rules restricting the generality of the original language definition, a reasonably simple parser can be generated. For instance, in [17], reserved words were introduced assuming that using them for user-defined identifiers was to be considered as bad practice anyway. By using true reserved words, `DECLARE` in figure 2.3 cannot refer to a function named “`DECLARE`”.

Figure 2.4 shows a spectacular example of of lexical ambiguity as supported by `PL/1`.

Figure 2.4 Another `PL/1` lexical problem

```
IF IF = THEN THEN THEN = IF ELSE ELSE = THEN
```

At this stage, it is clear that formalizing lexical analysis for `PL/1` requires a set of possibly more than one $Token$ at every position rather than a single $Token$ as we considered so far.

$Lexer_1$

$Lexer$

The restriction here is not as harsh as for $Lexer_0$, since we can now accept *multiple* interpretations at a given position, as long as they all have the same length. On the other hand, reducing to a shorter lexeme is forbidden, and so is skipping shorter white space In other words, after having recognized a word to be a token, it can be recognized as white space (by means of $Skippable$) of at least the same length, yielding the following word. In such cases, the coverages of the various $Tokens$ recognized at any position cannot intersect.

For instance, the following relative positioning of the $Token$s in $CurrentTokens$ is valid:

$\boxed{Tok_1}$

$\qquad\boxed{Tok_2}$

while this one would be invalid because of the overlap between Tok_1 and Tok_2:

$\boxed{Tok_1}$

$\quad\boxed{Tok_2}$

$$\forall\, x, y \in \operatorname{ran} CurrentTokens \mid x \neq y \bullet$$
$$\quad x.Position = y.Position \Rightarrow x.Length = y.Length \land x.Code \neq y.Code$$
$$\forall\, x, y \in \operatorname{ran} CurrentTokens \mid x \neq y \bullet$$
$$\quad x.Cover = y.Cover \lor x.Cover \cap y.Cover = \phi$$

2.1.2 FORTRAN

FORTRAN is not very well suited to the kind of parsers that can be generated automatically, because its lexical definition is ambiguous. There can be several ways of splitting the input of a FORTRAN source code into tokens, but only one of these ways is valid according to the syntax rules of the language. In other words, *separation of labour (as depicted in figure 2.2) does not apply*.

This issue has been addressed in , using the lookahead ability of the lexical analyzer generator they consider (namely, lex). It provides a lexical definition of the DO keyword as shown in figure 2.5.

Figure 2.5 Recognizing the DO keyword using lexical lookahead

```
DO/({letter} | {digit})* = ({letter} | {digit})*,
```

This lexical definition scans the input until the comma is found, in which case the first two letters "D" and "O" do refer to the DO keyword to indicate a loop. While apparently satisfactory, this scheme only works in this situation because FORTRAN happens to be a very simple language. Hence, scanning the input until the discriminating comma can be done using a regular expression. The prefix of a DO statement up to the comma can be matched by a single regular expression, but whatever follows DO is rejected using the rejection operator "/" for commodity purposes.

The pattern that follows the "/" operator is nothing else than syntactical information that describes the context in which "D" followed by "O" refers to the DO keyword. It takes advantage of the fact that the language is simple enough for syntactical construct to be recognizable using regular expressions only.

When dealing with more complex languages (for instance, if nested parentheses could be used between DO and the comma), this scheme would not be applicable, because the valid patterns that could follows DO cannot be expressed using regular expressions only. Recognizing nested parentheses requires some form of memory – to store the level of nesting – that can be handled by push-down automata (context-free languages) but not by deterministic automata (regular languages).

presents several lex-based solutions to FORTRAN's lexical problems in increasing difficulty order, and finally acknowledges that in such desperate cases, a hand-made lexical analyzer is probably much easier to write.

In the context of this work, we will consider two specific issues.

Figure 2.6 FORTRAN lexical problem

```
  IF IF THEN THEN = IF
```

The example listed in figure 2.6 shows clearly that the context-free[4] lexical analysis of FORTRAN is intrinsically ambiguous. The first IF is a keyword, the second one is

[4]Understand: without syntactical information nor abnormally long lookahead which would be similar to such syntactical information.

an identifier. This example is similar to what can be found in any language without reserved words, such as PL/1 , COOL and most natural languages.

Figure 2.7 FORTRAN lexical problem, continued

```
DO 10 I= 20,50
DO10I=20.50
```

The second example, presented on figure 2.7, is more FORTRAN-specific. The first of the statements is a loop that will branch to line 10 for I varying from 20 to 50, while the second is the assignment of the value 20.50 to a variable named DO10I, that will even be created implicitely if required. These two statements are lexically the same until the comma (or the dot in the second statement), since space characters are ignored.

So, some form of lookahead[5] is required before deciding whether the statement is an assigment or a loop statement. However, this requires more than just lexical lookahead, since it even has a deep effect on the mere *number* of lexemes:

In the first case, we have 7 lexemes (“DO”, “10”, “I”, “=”, “20”, “,”, “50”) while we only have 3 in the second case (“DO10I”, “=”, “20.50”). So, we will have to undo the parsing of a part of the statement, but we will also have to change the way the input string is broken into tokens, not just their meaning (code), but even their length and number.

Writing such an analyzer by hand requires special ad hoc techniques as well. In [41], Jean-Eric Waroquier mentions using a scanner, that performs simple analysis on a line[6] to remove ambiguities before proceeding with true scanning of the input with an *a priori* hint about the kind of statement the line represents.

2.1.3 COBOL

A COBOL program is divided in divisions and sections, that support different lexical rules. For instance, when declaring a variable VAL within a data division as depicted in figure 2.8 the token “X(9)” stands for an atomic picture clause that should be analyzed separately, to define the corresponding memory layout and data usage.

On the other hand, in a statement MOVE 0 TO X(9) such as listed in figure 2.8 in the procedure division, “X(9)” refers to the 9^{th} element of an array X, and should be divided into four lexemes, namely “X”, “(”, “9” and “)”. Early releases of the language definition [42] states that there every opening parenthesis in the procedure division should be preceeded by a newline, a space or another parenthesis[7]. However, several compilers [43] (probably using hand-crafted parsers) do not suffer from this restriction, and accept opening parenthesis without preceeding white space.

[5] At this preliminary stage, I should use the term *visibility* rather than lookahead which is attached to a very well-defined semantic in the compiler design literature.

[6] Since FORTRAN is a line-oriented language, that is, newline characters are significant and are used as statement separator.

[7] It is reasonable to believe that this restriction has been introduced to remove the ambiguity shown here.

Figure 2.8 `COBOL` lexical ambiguities

```
DATA DIVISION.
 ...
1 VAL PIC X(9).
 ...
PROCEDURE DIVISION.
 ...
   MOVE 0 TO X(9)
```

2.1.4 COOL

Beside having all the problems that apply to COBOL, COOL parsers have problems of their own, simply because recognizing several different COBOL dialects means that depending on the context, some words should be considered as reserved words or user-defined identifiers. We face the problem depicted in figures 2.6 and 2.4. A program that is accepted by a given compiler for a given dialect might have identifiers that are used as reserved words in another dialect. Hence, when one wishes to recognize several dialects within a single parser, this polymorphic ability must be taken into account.

$Lexer_2$, as defined here, allows the various paths that can describe the input to have different number of lexemes, of different lengths and positions. It is still restricted in the way it forbids two paths from each having a token at the same position, the same code and different length. $Lexer_3$ (See the current page) will lift all such restrictions.

This is to prevent a backtracking lexer while recognizing an identifier "FOOBAR" from retrying with all valid prefix identifiers: "FOOBA", "FOOB", "FOO", etc ...

On the other hand, when dealing with a situation as depicted in figure 2.7 in page 25, "DO10I" recognized as an identifier can backtrack to "DO" as a reserved word.

$Lexer_2$

$Lexer$

The restriction about lengths has disappeared: the only things that still cannot happen is having two tokens at the same position yielding the same $Code$ (See page 14).

$$\forall x, y \in \operatorname{ran} CurrentTokens \mid x \neq y \bullet$$
$$x.Position = y.Position \Rightarrow x.Code \neq y.Code$$

Finally, $Lexer_3$ is the most universal lexer possible, up to a point where it is barely usable due to a probably exponential behaviour in most cases.

$Lexer_3$

$Lexer$

$Lexer_3$ puts no restriction onto $CurrentTokens$.

2.2 Implementation notes

Formal specification are supposed to be as disconnected from the implementation as possible, in order to improve understanding. However, they are also supposed to serve as basis for the implementation, and while some concepts translate seamlessly into code, some others must be amended to cope with mundane issues such as performance or resource consumption.

The inheritance tree depicted in figure 2.1 is not implemented as a class hierarchy. The differences of behaviours between the various kinds of lexers are handled by ad hoc `IF` statements (See module *Lexical analyzer skeleton* on page 212).

Chapter 3

`lexyt` **user guide**

`lexyt` is a lexical analyzer generator, similar to `lex` in the principle, but which supports the various backtracking levels as described in chapter 2. This chapter describes how one can use `lexyt` to produce working lexical analyzer. The full source code of `lexyt` is described extensively in chapter 4.

3.1 First steps

`lexyt` takes a lexical specificationj in the form of an 8 bits ASCII file as input. It checks for its syntax and for some validity rules, and then generates two target files: $\mathrm{Y^AF_L}$ definition and implementation modules which, when compiled, will implement a usable *lexical analyzer* that matches the specification described in the input ASCII file.

A lexical specification is divided into two parts:

- A finite set of atomic *lexical definitions*
- Meta-commands, that control the way the generated lexical analyzer will behave.

`lexyt` can generate versatile lexical analyzers. They can *backtrack*, that is, the calling program can refuse the current token, and ask for another valid token *without advancing the input pointer*. More precisely stated, it can ask for another interpretation of the current token.

`lexyt` can generate lexical analyzers with 4 different levels of backtracking:

- Level 0, where the lexical analyzer always returns the token that matches the longest input string possible, and returns the first code that matches this string. This is exactly `lex` [40]'s behaviour.
- Level 1, where the lexical analyzer always returns the token that matches the longest input string possible, and can be asked to successively return all the codes that match the same longest string. In this case, if the longest string that can be matched at a given position is made of 8 characters, all alternative interpretations of that token that will be obtained by backtracking will be made of 8 characters.

- Level 2, where the lexical analyzer first returns the token that matches the longest input string possible, and can be asked to successively return all the codes that matches the string and its prefixes in decreasing length order, as long as a code that has been returned once on a given string length is never returned again for any of its prefixes. For instance, if the lexical analyzer has returned a code `Ident` for a string, it will never return that same code for a prefix of that string.

- Level 3, that can be qualified as ultimate backtracking, where no constraint is made. All valid interpretations of the current token will be returned successively, all valid prefixes with all valid codes will be considered. For virtuall all non-trivial lexical specifications, level 3 lexers have an exponential behaviour.

[1]Furthermore, lexical analyzers generated by `lexyt` will also have the ability to resynchronize onto a previously analyzed position, start forward again, possibly producing alternative interpretations of the input by backtracking if we are dealing with a lexer of backtracking level greater than 0.

3.2 Structure of the input file

`lexyt` recognizes a very simple input format:

- leading and trailing blanks are ignored – which does not imply that this must be the case for the language defined by the lexical specification

- blank lines, as well as lines that begin with a semicolon ";" or with a "#" character are ignored

- lines that start with a dollar character "$" are meta-commands

- all other lines should be lexical definitions.

3.2.1 Structure of a lexical definition

A lexical definition is made of a line starting with a valid YAFL identifier with a possible control character prefix, and a corresponding regular expression. For instance:

```
Comma     ,
```

indicates that the generated lexical analyzer will include a constant (which value is essentially irrelevant) named `Comma`. The attached value will be returned by the lexical analyzer when encountering a comma. When getting too complex, the regular expression attached to a lexical definitions can be split in smaller, more manageable pieces by using variables as described in 3.2.1.4.

The token can be prefixed by an underscore character "_" to indicate that the token should be recognized, and ignored without being transmitted to the calling program. This typically the case for comments and white spaces.

[1]For a more formal treatment of these various kinds of lexical analyzer, please refer to chapter 2.

```
_Ws             [ \n\t]+
_Comment        \-\-.*\n
```

The token name can also be used with an exclamation mark prefix to indicate that the generated lexical analyzer should return the standard code `ERROR` when encountering the token. It is then the calling program's responsibility to take any appropriate action. Typically, such lexical definitions are placed at the end of the specifications, to consider cases where none of the lexical definitions could be used.

```
Digit            [0-9]
Letter           [a-zA-Z]
Plus           \+
...
!Err             .
```

Unlike `lex`, `lexyt` does not support the inclusion of semantic actions attached to lexemes. The only action that `lexyt` can take when a given lexeme is recognized is to return the corresponding code. It is up to the calling program to take appropriate action when a given code is returned by the `lexyt`-generated lexer.

3.2.1.1 Regular expressions

The regular expressions as recognized by `lexyt` 's input format can include the following elements:

- Simple character, denoted by themselves or escaped using a backslash if the character can also be used as meta-character ("–", "+", "*", etc...)
- Character classes (see 3.2.1.2)
- Repetition postfix operators, "`*`" and "`+`", that denote resp. zero or more, and one or more repetitions of the argument. Beware, however, that in absence of explicit parentheses, `AB*` is understood as `A(B*)` and not as `(AB)*`.
- Parentheses can be used to group sub-expressions
- The empty string, denoted by an arobas "`@`".
- The disjunction operator, denoted by a vertical bar "|". To match "`YES`" or "`NO`" in a single regular expression, `(YES)|(NO)` must be used instead of `YES|NO`, which would match "`YESO`" and "`YENO`".

The last and most obvious regular operator is concatenation, which is implicit: when two regular expressions are put side to side, they are implicitely concatenated. For instance, the regular expression `YES` would be represented as `Y.E.S` if the dot represented explicit concatenation.

`lexyt` does not support the "?" postfix operator that denotes an optional sub-expression. The same functionality can be obtained using a form such as "`(A|@)`" which is equivalent to "`A?`".

3.2.1.2 Character classes

Character classes are defined within square brackets “[” and “]”.

```
[01234567890]
```

is strictly equivalent to

```
(0|1|2|3|4|5|6|7|8|9|0)
```

If the first character after the opening bracket is a “ˆ”, then the character class is negated, that is, it matches all characters that are *not* part of the character class specified without it. For instance,

```
[ˆ01234567890]
```

denotes all non-numeric characters.

Within character classes, ranges can be used using the dash character:

```
[0-9A-Za-z_]
```

denotes any character that is numeric, alphabetic or the underscore character. Beware that the dash character, unless escaped, is always understood as the range operator.

The dot is used as conveniency shortcut for any character that is not a newline character. It is equivalent to `[ˆ\n]`

3.2.1.3 Escaping special characters

All the special characters that are used to describe regular expression can be used for their literal value when preceeded by an escaping backslash character: `\.`, `\-`, `\[`, `\]`, `\(`, `\)`, `\{`, `\}`, `\*`, `\+`, `\|`, `\@` and even `\\`. The common `C` conventions for control characaters have also been used: `\n` denotes the newline character, while `\t` denotes the tab character.

The backslash character can also be used to denote non-printable characters. `lexyt` considers up to four[2] numerical characters following the backslash character as the decimal ordinal value of a character, as in `\200`, `\13`, `\10`. In the extreme case where one would want a pattern to be made of the character code 25 followed by the digit “`0`”, one would then need to separate the two subpatterns by an empty string code: `\25@0`.

3.2.1.4 Expanding variables

`lexyt` also supports the ability to reuse existing patterns within new patterns, by surrounding the inserted pattern name withing curly braces:

```
INumber    [0-9]+
RNumber    {INumber}\.{INumber}
```

This is strictly equivalent to

[2] Four digits codes are supported for lexers transitions made on character sets larger than 8 bits.

```
INumber   [0-9]+
RNumber   [0-9]+\.[0-9]+
```

One can also reuse parts of regular expressions without having even defined them as plain patterns (See below).

3.2.2 Available meta-commands

3.2.2.1 $VAR

The `$VAR` command allows the user to define a new variable that denotes a regular expression, without defining it as a token, that is, in such a way that the generated lexical analyzer will never return it as a valid token. Such a variable is merely used as a reusable regular expression component:

```
$VAR  PlusMinus (\+|\-|@)
$VAR  Digits      ([0-9]+)

INumber   {PlusMinus}{Digits}
RNumber   {PlusMinus}{Digits}\.{Digits}
```

`lexyt` pastes variables using string substitution only, not as logical regular expressions. Hence, if the variable definition's regular expression is not enclosed in parentheses (`$VAR  PlusMinus \+|\-|@` as opposed to `$VAR  PlusMinus (\+|\-|@)`), pasting it might cause undesired behaviour.

For instance,

```
$VAR  PlusMinus \+|\-|@
$VAR  Digits      ([0-9]+)

Signs     {PlusMinus}+{Digits}
```

will define `Signs` as `\+|\-|@+([0-9]+)` rather than `(\+|\-|@)+([0-9]+)` which was probably the desired result.

3.2.2.2 $IGNORECASE and $NOIGNORECASE

When dealing with case-insensitive lexical definitions, we can always use the lengthy explicit method:

```
Begin       (b|B)(e|E)(g|G)(i|I)(n|N)
```

Admittedly, this would soon prove cumbersome and not very productive. `lexyt` supports the `$IGNORECASE` meta-command, that can be used to indicate that all character transitions should be understood in a case-independent way:

```
$IGNORECASE
Begin       begin
End         end
```

The default behaviour for `lexyt`-generated lexical analyzers is to be case-dependent. It is also possible to mix behaviours by using `$NOIGNORECASE` to turn on case dependency back again.

3.2.2.3 $BACKTRACKLEVEL

The meta-command `$BACKTRACKLEVEL` can be used to indicate the level of backtracking that must be supported (see 3.1).

```
$BACKTRACKLEVEL 3
```

A “`D`” can be specified instead of a numeric value, as in:

```
$BACKTRACKLEVEL D
```

to indicate that the backtracking level of the generated parser is dynamic, that is, that it must be set by the enclosing application (See method *YLexicalAnalyzer.YLexicalAnalyzer.SetBackTrackLevel* on page 187). This feature induces a significant performance penalty when compared with lexers that are generated with a statically defined backtracking level.

The backtracking level is a global characteristic of the lexer. If several $BACKTRACKLEVEL meta-commands are listed in the source, only the last one is taken into consideration.

3.2.2.4 $CLASS

The `$CLASS` meta-command can be used to define a new lexical definition based on an existing set of lexical definitions.

Figure 3.1 Sample lexical specification

```
While      while
Do         do
For        for
If         if
Else       else
Case       case
Switch     switch
```

For instance, consider the lexical specification given in figure 3.1. This specification describes a subset of C’s reserved words. In a level 1 (or above) lexical analyzer, it might be useful to define a token as being *any* reserved word. In a level 0 lexical analyzer, this would not make any sense, since the analyzer does not even have the ability to ask for another, more general incarnation of the current token. Hence, when recognizing that a token is, let say, a `struct` keyword, it will not give any alternate significance for the current token.

It is of course possible to define such a `ReservedWord` token, as in figure 3.2.

Figure 3.2 Lexical specification with verbatim disjunction of all reserved words

```
$BACKTRACKLEVEL 1
While      while
Do         do
For        for
If         if
Else       else
Case       case
Switch     switch
ReservedWord (while)|(do)|(for)|(if)|(else)|(case)...
```

This approach, while workable, is awkward at best. In languages with over a dozen reserved words, it is totally unreasonable. A similar result with a much more compact notation would be achieved using the `$CLASS` meta-command (see figure 3.3).

Figure 3.3 Lexical specification using the `$CLASS` keyword

```
$BACKTRACKLEVEL 1
While      while
Do         do
For        for
If         if
Else       else
Case       case
Return     return
Union     union
Switch     switch
$CLASS ReservedWord While-Switch
```

A `$CLASS` meta-command first indicates the name of the newly created token, followed by a list of tokens or token ranges. For instance, it would be acceptable, if not sensible, to define a token as:

```
$CLASS NonSense While-For Else-Case Union
```

This command creates a new token named `NonSense`, for which the lexical definition is made of the disjunction of the lexical definitions attached to the set of tokens denoted by `While-For Else-Case Union`, that is: “`While`”, “`Do`”, “`For`”, “`Else`”, “`Case`” and “`Union`”.

3.2.2.5 $EXCLUDE

The `$EXCLUDE` meta-command can be used to control backtracking, by excluding some some alternate interpretations of the input which are valid if based on the lexical specification only. For instance, consider the specification depicted in figure 3.4.

Figure 3.4 Lexical specification using the `$EXCLUDE` keyword

```
$BACKTRACKLEVEL 2
End          END
EndIf        END\-IF
Ident        [a-zA-Z][a-zA-Z0-9\-]*

$EXCLUDE EndIf End Ident
$EXCLUDE End Ident
```

`$EXCLUDE EndIf End Ident` indicates that after having recognized a lexeme with code `EndIf`, backtracking must not return `End` or `Ident`, even if their lexical definition might allow it. In other words, this means that `END-IF` cannot be used as a valid identifier, and that it cannot be split into a leading `END` and some other trailing lexemes. It is a logical cut but somehow restricted in terms of the alternate paths it discards explicitely.

3.3 Capturing comments

Lexemes marked with a leading underscore "_" are ignored in the sense that they are recognized in the input, but the calling system is not notified whenever one of such comments has been skipped. In some applications that need the comments (pretty printing, for instance), one can ask the generated lexer to notify whenever one of these comments has been recognized and skipped.

In order to do so, one must:

- Call the `SetKeepComments` method (See page 189) to give to the generated lexer the list of the codes that ought to be considered as useful comments.
- Redefine the `Skip` method (See page 188) as defined for the lexer to take some action if one of these useful comments has been recognized.

If the `Skip` method is not redefined, its default behaviour will be to create instances of the `Comment` class (See page 206) and to attach them to the lexer. These comments can then be consulted using the `GetComments` (See page 189) method.

3.4 Command-line options

`lexyt` can be invoked by using a command line of the form:

```
lexyt {options}* InputFile ModuleName
```

`InputFile` denotes a file that contains a – hopefully – valid lexical specification as described above. `ModuleName` denotes the module name of the generated lexer. This module name can be any valid YAFL identifier, `lexyt` will produce files which names match YAFL's rules for the correspondance between module and file names. For instance, when generating a lexer in a module named `CobolLexer`, `lexyt` produces a definition module "`cobollex.dy`" and an implementation module "`cobollex.yy`".

Valid options recognized by `lexyt` are listed in table 3.1.

Table 3.1: Command-line option recognized by `lexyt`

-h or ?	Display on-line help information about `lexyt`'s usage
-v	Set verbose mode, where some information is displayed about the lexical specification and the generated lexer. Otherwise, `lexyt` works silently
-p	Set profiling mode, where YAFL 's facilities for profiling are used to generate a report file describing how many times each statement is executed. Please note that this profiling mode refers to `lexyt` and not the the generated lexer, that can be profiled separately
-i	Ignore case, equivalent to the `$IGNORECASE` meta-command in the lexical specification. If the meta-command is found in the source file (See page 33), it overrides the command-line option
-b*num*	Set the generated lexer's backtracking level, similarly to the `$BACKTRACKLEVEL` meta-command. If this meta command occurs in the source file, it overrides the command-line option
-bd	Generate a lexer with a dynamic backtracking level
-k	Generate ancillary methods, each one controlling the behaviour of the lexer when in a given state. Otherwise, `lexyt` produces the entire lexer in nested `CASE` statements, that can prove difficult to compile in practice. The YAFL compiler can compile even the largest generated lexer flawlessly, but some of the optimisations performed by the `C` compiler cause the machine to hang when given the huge `C` function generated by the YAFL compiler out of these large nested `CASE` statements.
-y*fname*	Instruct `lexyt` to produce an equivalent `yacc` grammar in a file named *fname* instead of producing a lexer. This `yacc` grammar will use atomic characters as lexemes.
-m*num*	Instruct `lexyt` to use up to *num* bytes of dynamic memory. This option might prove useful when producing lexers based on very large and/or very complex lexical specifications.
-w*num*	Use *num* as word length, valid values are 8, 4, 2, 1.
-t*num*	Accept *num* different codes for atomic characters. The default value for this option is 256. This option is meant to be used to produce lexers that deal with character sets larger made of more than 256 elements. *While recognized and documented, this option is not yet implemented, and using it will have no effect on the generated lexer.*
-f*fname*	Produce the transition table for the lexer in the external file *fname* instead of producing a compiled lexer, where the transition tables are hard wired in the generated code. *While recognized and documented, this option is not yet implemented, and using it will have no effect on the generated lexer.*

Chapter 4

`lexyt` source code

The following modules implement the lexical analyzer generator. The strategies listed in chapter 2 are described extensively here. Of course, this chapter also includes some mundane operations of far less theoretical interest but this is the price to pay to go through a real-world implementation instead of being restricted to some theoretical but useless study.

4.1 The internal structure of `lexyt`

`lexyt` takes an input ASCII file that contains lexical definitions, and generates the YAFL source code for a lexical analyzer that will recognize these tokens.

In order to achieve this goal, `lexyt` uses several intermediate representations as shown in figure 4.1.

1. Each lexical definition is tokenized, that is, represented as a sequence of codified tokens, indicating simple characters or metacharacters ($*$, $+$, $-$, etc...). This simple operation is performed by the `Codify` method, implemented as part of the `ExpressionParser` class (See page 100). Escaped characters (that is, the difference between a plain "*" and "*") are handled at this level. This step also takes care of variable pasting.

2. This sequence of codified integers is parsed to produce a parse tree that represents the regular expression. This pass is responsible for taking operator precedence into account, parentheses handling, character class decoding. It is implemented by the `Parse` method of the `ExpressionParser` class (See page 117), and the resulting expression trees are described in the the `Expression` module (See page 80).

3. This tree-based expression is then turned into a non-deterministic finite state automaton (in short, NFA). This construction is commonly referred to as *Thompson's construction* . The NFA is described in the `Node` module (See page 134).

4. All the NFA's of the specification are combined to form a single NFA (See method *Token.LexSpecification.BuildNFAAutomaton* on page 65), which is turned into a DFA using a well-known algorithm listed in , for which some theoretical background can be found in [29]. DFA states are represented by sets of NFA states, described in the `NodeSet` module (See page 149), and the combination of these DFA states is a DFA, described in the `DfaAutomaton` module (See page 123).

5. Finally, using a skeleton file (See page 212), the YAFL code for the lexical analyzer is generated using the DFA. The skeleton file is just a text file where the common parts of all lexical analyzers are described, and where codes are used to indicate that `lexyt` should paste code that depends on the actual lexical analyzer, the definition of its tokens and their properties.

It might be argued that a simpler and faster implementation could have been achieved using fewer passes. In practice, the performance issue is largely irrelevant. Over 90% of the processing time is spent when converting the NFA into the DFA. Optimizing the rest would have a negligible if even noticable impact on the resulting performance. The only way one could have achieved much better generation performance would have been to use an algorithm similar to `lex`'s that goes directly from regular expressions to DFA's.

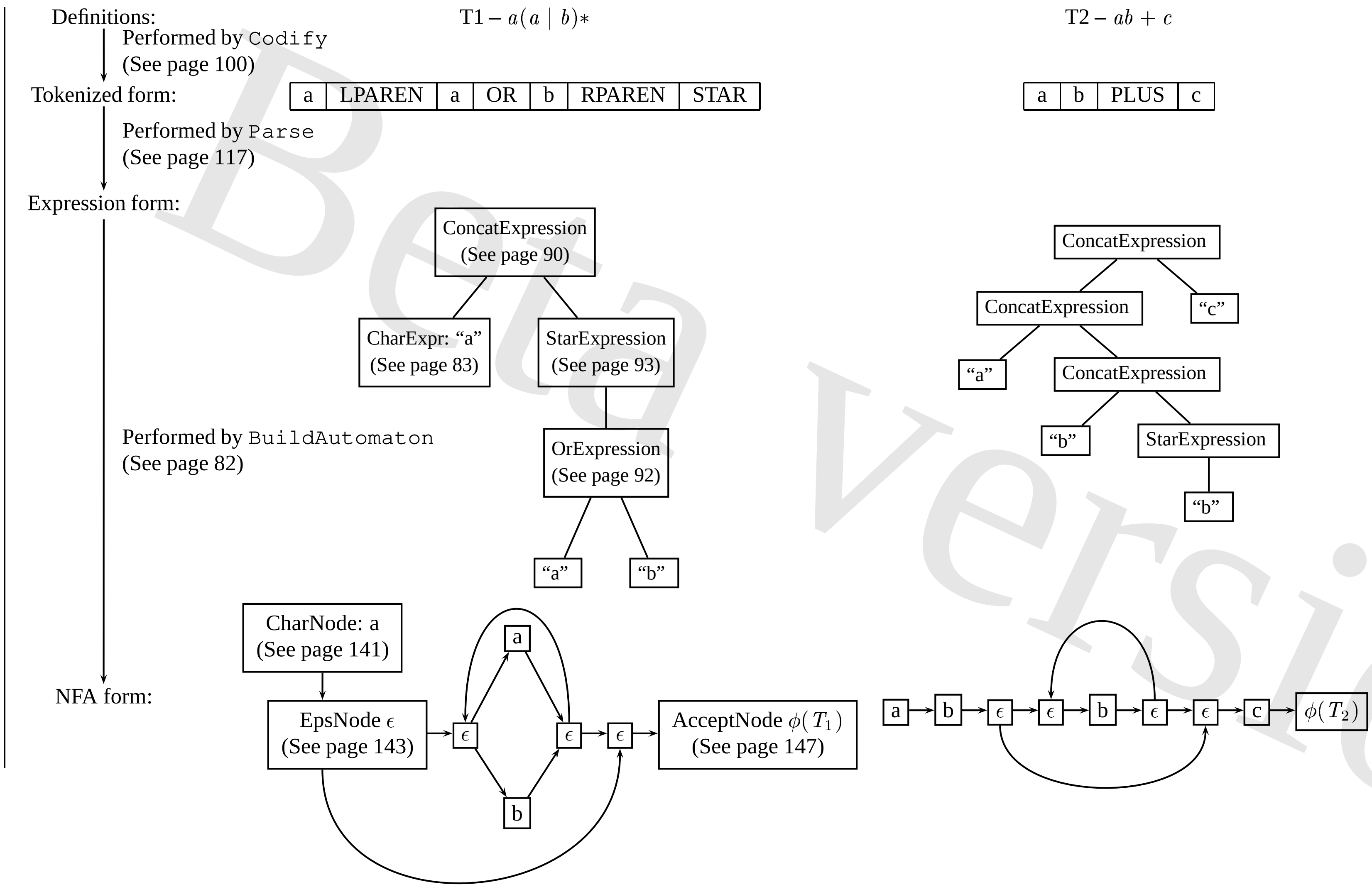
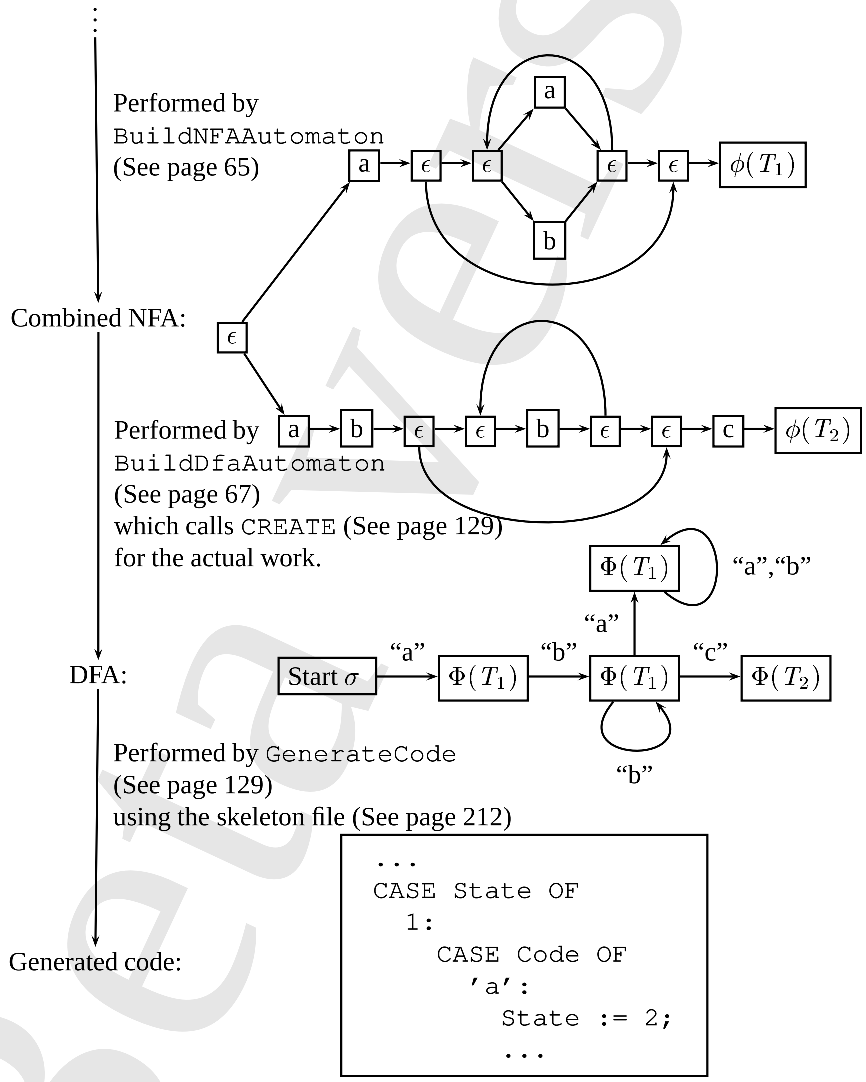


Figure 4.1 The intermediate representations of the lexer

Figure 4.2 The intermediate representations of the lexer, continued . . .



On the other hand, formally splitting the total processing into phases as described above has given us more freedom and flexibility to handle versatile features that were required to handle the complex lexical specification we were considering, such as variables, token classes, transitions of various lengths, the generation of an equivalent `yacc` grammar, etc . . .

Using the NFA, we implemented the token classes without making the number of NFA states explode. Without explicitely representing the NFA, implementing token classes would still have been possible, but my guess is that it would have been far more difficult, and that the resulting implementation would have been awkward and inefficient.

I also believe that our simplistic and systematic approach makes the source code far easier to read. Of course, this is up to you, beloved reader, to judge.

4.2 `lexyt` main module

4.3 YLex

The main module `YLex` manages such mundane work as reading the options on the command line, reading the input file, ensuring that the input contains no error, and call the code generation pass to actually produce the lexical analyzer.

⟨*Definition* ⟩≡

```
DEFINITION MODULE YLex;

   ⟨Phony import clause 46⟩

   CLASS LexYt;

      CONST
         Version = ''3.19'';  -- Version is a character
                              -- string. If one requires the ability
                              -- to perform floating point arithmetics
                              -- on this value, we might reconsider
                              -- this constant's type...

      REDEFINE METHOD CREATE;
   END LexYt;

END YLex;
```

The following import clause serves no other purpose than making sure that the `YLexicalAnalyzer` module (See page 183) is compiled whenever `lexyt` is rebuilt. This module is not required to run `lexyt` , but including it in this import list serves a version checking purpose. Whenever `lexyt` is compiled, `YLexicalAnalyzer` is checked for as well. In practice, the global optimiser [23] will remove all dead code anyway so importing this module induces no significant space penalty.

46 ⟨*Phony import clause* 46⟩≡

```
IMPORT YLexicalAnalyzer;
```

⟨*Implementation* ⟩≡

```
IMPLEMENTATION MODULE YLex;

FROM Conversions IMPORT IntConversions;
IMPORT DfaAutomaton;
IMPORT YLexCfg;
FROM Node IMPORT NodeDictionary;
FROM Token IMPORT Token, LexSpecification;
FROM Streams IMPORT StdOut, StdErr;
IMPORT String;
IMPORT SYSTEM;
IMPORT Profile;
```

⟨*Implementation* ⟩+≡

```
  CLASS LexYt;

    VAR
      TokList: LexSpecification;

    ⟨Help method 50d⟩
    ⟨CREATE method 47a⟩
  END LexYt;

END YLex;
```

The `CREATE` predefined method attached to the `YLex` class parses the command line parameters, sets the options accordingly, and if everything happened flawlessly (See method *YLexCfg.YLexCfg.Error* on page 181), actually generates the lexical analyzer.

47a ⟨`CREATE` *method* 47a⟩≡
```
REDEFINE METHOD CREATE;
   VAR
      ⟨CREATE local vars 48b⟩
   BEGIN
   IF SYSTEM.ArgCount = 0 THEN
      YLexCfg.SetOption (‘‘H”);
      END;
   WHILE (ArgNr < SYSTEM.ArgCount) ∧ ¬ YLexCfg.Error DO
      a ← SYSTEM.Argument(ArgNr);
      ⟨Handle option a 48e⟩
      ArgNr ← ArgNr + 1;
      END;
```

If help information is required by the user, it is displayed (See method *YLex.LexYt.Help* on page 50) and further processing is discarded.

47b ⟨`CREATE` *method* 47a⟩+≡
```
IF YLexCfg.Help THEN
   Help;
   YLexCfg.SetError (TRUE);
 ELSIF ¬ YLexCfg.Error THEN
   ⟨Check for errors 49c⟩
   END;
```

If we got as far as having only valid command-line options, we can start analyzing the input file, and if everything goes right, start producing the proper automata and ultimately, the target module.

47c ⟨`CREATE` *method* 47a⟩+≡
```
IF ¬ YLexCfg.Error THEN
   ⟨Read the token list 49d⟩
   IF ¬ YLexCfg.Error THEN
      ⟨Check for valid ModuleName 50c⟩
      IF YLexCfg.Verbose THEN
         StdOut.WriteInt (NodeDictionary.Count, 0);
         StdOut.WriteLine (‘‘ NFA states”);
         END;
```

If the proper command-line option has been set (See table 3.1 on page 38), `lexyt` generates an equivalent `yacc` grammar (See method *Token.LexSpecification.GenerateYacc* on page 77) instead of actually producing a lexer.

48a ⟨CREATE *method* 47a⟩+≡
```
      IF YLexCfg.GenerateYaccFName ≠ Λ THEN
        TokList.GenerateYacc (YLexCfg.GenerateYaccFName);
       ELSE
        ⟨Build the DFA automaton 50a⟩
        ⟨Generate code based on the DFA automaton 50b⟩
        END;
      END;
    END;
  END CREATE;
```

`ArgNr` denotes the index when scanning the command line arguments.

48b ⟨CREATE *local vars* 48b⟩≡
```
  ArgNr: INTEGER;
```

`FName` denotes the input file name where the lexical definitions will be found, `ModuleName` denotes the name of the target module where the lexical analyzer will be generated. It is a complete name that can be longer than 8 characters and contain upper and lower case characters. It will be normalized before actually creating the target file name. `a` is just a temporary variable.

48c ⟨CREATE *local vars* 48b⟩+≡
```
  a, FName, ModuleName: ARRAY OF CHAR;
```

Well, just another local variable...

48d ⟨CREATE *local vars* 48b⟩+≡
```
  MemSize: INTEGER;
```

48e ⟨*Handle option* a 48e⟩≡
```
  ASSERT a ≠ Λ;
  ASSERT a.SIZE > 0;
  CASE a0 OF
    ''?'':
      YLexCfg.SetOption (''H'');
      END;
    ''–'':
      ⟨Handle command-line option a 49a⟩
      END;
   ELSE
    IF FName = Λ THEN
      FName ← a;
     ELSE
      ModuleName ← a;
      END;
    END;
```

Command-line options must be at least two characters long, since they must always include a leading dash "-". The memory size option `-m` is handled specifically, while all other options are handled back to the `YLexCfg` (See page 175) once class that centralizes the global configuration and status information.

49a ⟨*Handle command-line option* a 49a⟩≡

```
a ← String.UpperCase (a);
IF (a.SIZE < 2) THEN
  StdErr.WriteLine (“Syntax error in command line:” + a);
  YLexCfg.SetError (TRUE);
 ELSIF SYSTEM.UCASE(a₁) = ‘‘M’’ THEN
  ⟨Handle memory size option −m 49b⟩
 ELSE
  YLexCfg.SetOption (a.SLICE (1, a.SIZE - 1));
  END;
```

49b ⟨*Handle memory size option* −m 49b⟩≡

```
MemSize ← IntConversions.StringToInt (a.SLICE(2, a.SIZE - 2));
IF MemSize > 1024 THEN
  SYSTEM.SetMaxMem (MemSize);
  END;
```

The following code makes sure that both an input file and a target module name have been specified on the command line.

49c ⟨*Check for errors* 49c⟩≡

```
IF FName = Λ THEN
  StdErr.WriteLine (“No input file specified”);
  YLexCfg.SetError (TRUE);
 ELSIF ModuleName = Λ THEN
  StdErr.WriteLine (“No target module name specified”);
  YLexCfg.SetError (TRUE);
  END;
```

The following code fragment basically creates a token list and parses an input file into it. Any error condition will be detected using the `YLexCfg` module (See page 175).

49d ⟨*Read the token list* 49d⟩≡

```
⟨Check for valid ModuleName 50c⟩
TokList.CREATE;
TokList.ReadFile (FName);
```

`Dfa` denotes the automaton (See class *DfaAutomaton.DfaAutomaton* on page 123) that will be created based on the lexical definitions that are found in the input. It will be created if no error has been found.

49e ⟨`CREATE` *local vars* 48b⟩+≡

```
Dfa: DfaAutomaton;
```

Well, when all these tokens have been read successfully from the lexical specification, `BuildDfaAutomaton` (See page 67) goes through all the phases shown in figures 4.1 and 4.2 in pages 43 and 44 to build the DFA automaton.

50a ⟨*Build the DFA automaton* 50a⟩≡

```
Dfa ← TokList.BuildDfaAutomaton;
```

Then, this DFA is used to generate an output file that must be a valid YAFL source module (See class *DfaAutomaton.DfaAutomaton* on page 123).

50b ⟨*Generate code based on the DFA automaton* 50b⟩≡

```
IF ¬ Dfa.GenerateCode (ModuleName) THEN
  StdErr.WriteLine (‘‘Error while generating the target file”);
  END;
IF YLexCfg.Verbose THEN
  Dfa.DumpStatistics;
  END;
```

50c ⟨*Check for valid* `ModuleName` 50c⟩≡

```
ASSERT ModuleName ≠ Λ;
ASSERT ModuleName.SIZE > 0;
```

4.3.1 Mundane stuff

The `Help` method merely displays some user-readable help information on the standard output.

50d ⟨`Help` *method* 50d⟩≡

```
METHOD Help;

  METHOD l(a: ARRAY OF CHAR);
    BEGIN
    StdOut.WriteLine (a);
    END l;
```

The help information is hard-coded in the source rather than stored in an external file to avoid discrepancies between the version of an executable image and a possibly unrelated documentation file.

51 ⟨Help *method* 50d⟩+≡

```
BEGIN
l ("LexYt - Yafl Lexical Analyzer - Version " + Version);
StdOut.WriteLn;
l ("Syntax: lexyt {options}* inputfile ModuleName");
StdOut.WriteLn;
l ("inputfile: list of token descriptions using regular " +
             "expression notation");
StdOut.WriteLn;
l ("ModuleName: a valid Yafl MODULE or CLASS name used " +
                  "to generate the analyzer");
l ("            from 2 skeleton files dskel.lex and " +
                  "iskel.lex");
StdOut.WriteLn;
```

52 ⟨Help *method* 50d⟩+≡

```
  l (‘‘Possible options are: -h or ? Display this help”);
  l (‘‘                      -v verbose”);
  l (‘‘                      -p profile mode”);
  l (‘‘                      -i ignore case in identifier”);
  l (‘‘                      -k generate ancillary methods instead ”);
  l (‘‘                         of a huge CASE statement”);
  l (‘‘                      -b<num> generate a lexical analyzer”);
  l (‘‘                       of backtracking level <num>”);
  l (‘‘                      -bd generate a lexical analyzer”);
  l (‘‘                       with a dynamic backtracking level”);
  l (‘‘                      -m<num> uses <num> bytes memory”);
  l (‘‘                          These two optimizations refer”);
  l (‘‘                          to the lexer generation process,”);
  l (‘‘                          not to the lexer itself”);
  l (‘‘                      -w<num> use <num> as word length”);
  l (‘‘                          valid values are 8, 4, 2, 1”);
  l (‘‘                      -t<num> use <num> as the number of” +
                                            ‘‘ different”);
  l (‘‘                          token codes (default is 256)”);
  l (‘‘                        -y<fname> generate equivalent yacc ” +
                               ‘‘parser in <fname>”);
  l (‘‘                        -f<fname> generate interpreted ” +
                                ‘‘lexer in <fname>”);
  StdOut.WriteLn;
  l (‘‘Recognized keywords are:”);
  l (‘‘     $VAR $CLASS $BACKTRACKLEVEL $IGNORECASE” +
         ‘‘ $NOIGNORECASE $EXCLUDE”);
  END Help;
```

4.4 Token

The `Token` module describes `lexyt`'s input format, and provides methods to produce the intermediate representation to generate the lexical analyzer. It defines two classes:

- The `Token` class, that denotes a single lexical definition.
- The `LexSpecification` class, that denotes an entire lexical specification, made of a list of tokens and additional information, such as variables.

⟨*Definition* ⟩≡

```
DEFINITION MODULE Token;

FROM Automaton IMPORT Automaton;
FROM DfaAutomaton IMPORT DfaAutomaton;
FROM List IMPORT StringList, StringElement;
FROM Node IMPORT Node;
FROM Streams IMPORT InputStream, OutputStream;

  ⟨Token class definition 54a⟩
  ⟨LexSpecification class definition 63a⟩

END Token;
```

⟨*Implementation* ⟩≡

```
IMPLEMENTATION MODULE Token;

IMPORT YLexCfg;
FROM Conversions IMPORT IntConversions;
FROM List IMPORT List;
FROM Node IMPORT AcceptNode, EpsNode, CharNode, CharSetNode;
FROM NodeSet IMPORT NodeSet;
FROM Expression IMPORT ExpressionParser, Expression;
FROM Streams IMPORT StdErr, StdOut;
IMPORT String;
IMPORT SYSTEM;

  ⟨Token class implementation 54b⟩
  ⟨Variable private class implementation 79⟩
  ⟨LexSpecification class implementation 63b⟩

END Token;
```

4.4.1 The `Token` class

The `Token` class inherits from `StringElement` as provided by YAFL's standard library so that `StringList` instances of `Tokens` can be searched by name. For instance, `LexSpecification` (See page 63) inherits from `StringList`, so that existing searching primitives defined for `StringList` can be used to look for a `Token` within a `LexSpecification`.

54a ⟨Token *class definition* 54a⟩≡

```
CLASS Token;
  INHERITS StringElement;

  ⟨Token creation method definition 60c⟩
  ⟨Trivial consultation methods, definition 59d⟩
  ⟨SymbolName method definition 58c⟩
  ⟨Code generation method definitions 57b⟩
  ⟨Handling exclusion, definition 59b⟩

END Token;
```

54b ⟨Token *class implementation* 54b⟩≡

```
CLASS Token;
  INHERITS StringElement;

  ⟨Token attributes 55a⟩
  ⟨BuildAutomaton method implementation 55f⟩
  ⟨Token creation method implementation 61a⟩
  ⟨ParseDefinition method, that parses the lexical definition 56d⟩
  ⟨MakeClass method implementation, to create large disjunctions 57a⟩

  ⟨Code generation method implementations 58a⟩
  ⟨Handling exclusion, implementation 59c⟩

  ⟨Trivial consultation methods, implementation 60a⟩

  ⟨SymbolName method implementation 59a⟩

  ⟨GenerateYacc method implementation 62⟩

END Token;
```

`TheDefinition` denotes the textual description of the token in the lexical specification. If a token is defined by:

```
Ident     [a-zA-Z][a-zA-Z0-9_]
```

its attached `TheDefinition` wil hold `"[a-zA-Z][a-zA-Z0-9_]"`. It is retained as an attribute for reusability purposes: when a token is used to define another token, its definition is pasted in that other token's definition.

In a future release, it might also serve for error reporting purposes, giving the ability to indicate where an error occurred in the source code.

55a ⟨`Token` *attributes* 55a⟩≡
```
VAR
  TheDefinition: ARRAY OF CHAR;
```

`TheAutomaton` denotes the corresponding NFA (See page 119), that is built using the `BuildAutomaton` method (See the current page).

55b ⟨`Token` *attributes* 55a⟩+≡
```
TheAutomaton: Automaton;
```

`TheExpr` denotes the token's lexical definition as an intermediate regular expression, using a tree-like representation (See module *Expression* on page 80), while `TheNr` denotes its sequence number within the lexical specification.

55c ⟨`Token` *attributes* 55a⟩+≡
```
TheExpr: Expression;
TheNr: INTEGER;
```

`SkipFlag` and `ErrorFlag` indicate whether `THIS` denotes a token that must be resp. skipped (typically, white spaces) or considered as an error (characters that are not part of any token's definition).

55d ⟨`Token` *attributes* 55a⟩+≡
```
  SkipFlag,
  ErrorFlag: BOOLEAN;
```

`TheSpec` denotes its enclosing lexical specification.

55e ⟨`Token` *attributes* 55a⟩+≡
```
TheSpec: LexSpecification;
```

The `BuildAutomaton` method builds the NFA attached to the token denoted by `THIS`, and assigns it to `TheAutomaton` (See the current page).

55f ⟨`BuildAutomaton` *method implementation* 55f⟩≡
```
METHOD BuildAutomaton;
  VAR
    a: Automaton;
    AccNode: AcceptNode;
  BEGIN
```

It does so in three steps. First, it builds the automaton attached to the expression defined as part of the token

55g ⟨`BuildAutomaton` *method implementation* 55f⟩+≡
```
a ← TheExpr.BuildAutomaton;
```

Theending node of the automaton must be the last one, that is, it cannot be connected to anything else.

56a ⟨BuildAutomaton *method implementation* 55f⟩+≡
```
ASSERT a.Last.Connected = Λ;
```

Then, it creates and appends a new accepting node (See class *Node.AcceptNode* on page 147) that accepts the token denoted by THIS to that automaton

56b ⟨BuildAutomaton *method implementation* 55f⟩+≡
```
AccNode.CREATE (THIS);
a.Last.Connect(AccNode);
```

Finally, it builds a new automaton including this new state, and assigns it to TheAutomaton.

56c ⟨BuildAutomaton *method implementation* 55f⟩+≡
```
TheAutomaton.CREATE (a.First, AccNode);
END BuildAutomaton;
```

The ParseDefinition private method takes a regular expression as a character string, builds the corresponding tree made of instances of the Expression class (See page 81), and if no error has been found, calls the BuildAutomaton method so that a NFA corresponding to this regular expression gets attached to THIS (See figure 4.1 on page 43).

56d ⟨ParseDefinition *method, that parses the lexical definition* 56d⟩≡
```
METHOD ParseDefinition (Definition: ARRAY OF CHAR);
  BEGIN
  ASSERT Definition ≠ Λ;
  TheDefinition ← Definition;
  TheExpr ← ExpressionParser.Parse(Definition, TheSpec);
```

Parsing the definition can cause errors, such as mismatched parentheses. If not such error has occurred, the automaton can be built (See method *Token.Token.BuildAutomaton* on the preceding page) .

56e ⟨ParseDefinition *method, that parses the lexical definition* 56d⟩+≡
```
IF ¬ YLexCfg.Error THEN
  BuildAutomaton;
  END;
END ParseDefinition;
```

Besides being created by parsing a lexical specification, new tokens can also be created by means of a higher-level command `$CLASS`, that indicates that a given token is the conjunction of a potentially large number of existing tokens (see figure 3.3 on page 35).

This feature is implemented by the `MakeClass` method, that defines `THIS` as being made of the disjunction of an array of other tokens. However, this method does nothing more than building an array with all the `Expressions` from which the disjunction must be computed; and to call `MakeTokenClass` (See page 117).

57a ⟨`MakeClass` *method implementation, to create large disjunctions* 57a⟩≡

```
METHOD MakeClass (Arr: ARRAY OF Token);
  VAR
    ExprArray: ARRAY OF Expression;
  BEGIN
  ExprArray.CREATE (Arr.SIZE);
  FOR i ∈ 0 .. Arr.SIZE - 1 DO
    ExprArray_i ← Arr_i.TheExpr;
    END;
  TheExpr ← ExpressionParser.MakeTokenClass (ExprArray);
  IF ¬ YLexCfg.Error THEN
    BuildAutomaton;
    END;
  END MakeClass;
```

Let's now consider the code generation methods attached to tokens, that is, methods that will be used when generating the target file.

57b ⟨*Code generation method definitions* 57b⟩≡

```
METHOD GenerateConstDecl (Output: OutputStream);
METHOD GenerateTokenImage (Output: OutputStream);
```

The `GenerateConstDecl` method is called for all the tokens attached to a lexical specification. It outputs constant declarations on a definition module so that external systems can refer to the tokens in a symbolic way. The `GenerateConstDecl` method generates a valid YAFL constant declaration for the token denoted by `THIS`. This constant declaration is computed by adding the number of the token to a predefined constant `FIRSTTOKEN`.

For instance, if `Ident` is the fourth lexical definition in the specification:

```
Ident      [a-zA-Z_][a-zA-Z0-9_]*
```

calling the `GenerateConstDecl` method on the corresponding `Token` instance will produce the following output:

```
Ident = FIRSTTOKEN + 4;
```

GenerateConstDecl is based on the Name method (See page 60).

58a ⟨*Code generation method implementations* 58a⟩≡

```
METHOD GenerateConstDecl (Output: OutputStream);
  BEGIN
  IF Name ≠ Λ THEN
    Output.WriteString (“      ” + Name + “ = ” );
    IF Skip THEN
      Output.WriteString (“SKIPOFFSET + ”);
     ELSE
      Output.WriteString (“FIRSTTOKEN +”);
      END;
    Output.WriteInt (TheNr, 0);
    Output.WriteLine (“;”);
    END;
  END GenerateConstDecl;
```

Similarly, the GenerateTokenImage method generates an entry in a CASE construct, that, depending on an integer value that should denote a token, returns its name in a human readable character string form.

For instance, given the Ident exemple presented above, lexyt would generate:

```
Ident: RESULT := "Ident"; END;
```

58b ⟨*Code generation method implementations* 58a⟩+≡

```
METHOD GenerateTokenImage (Output: OutputStream);
  BEGIN
  IF Name ≠ Λ THEN
    Output.WriteLine (Name + “: RESULT := ’” + Name + “’; END;”);
    END;
  END GenerateTokenImage;
```

The SymbolName method returns a character string that denotes the name of the YAFL symbolic constant attached to a token. This symbolic constant will be returned by the lexical analyzer when the token denoted by THIS will have been recognized.

It consider the case where a standard *ERROR* code should be returned instead of any user-defined code.

Otherwise, the name of the constant as declared by the GenerateConstDecl method described above is returned.

58c ⟨SymbolName *method definition* 58c⟩≡

```
METHOD SymbolName: ARRAY OF CHAR;
```

59a ⟨`SymbolName` *method implementation* 59a⟩≡

```
METHOD SymbolName: ARRAY OF CHAR;
  BEGIN
  IF Error THEN
    RESULT ← ‘‘ERROR”;
   ELSE
    RESULT ← Name;
    END;
  END SymbolName;
```

Every token t has a possible empty list of exclusion tokens attached. When a token t' is in the exclusion list of t, it means that after that a lexer has recognized a t at some position of the input, it cannot backtrack onto a t' unless the position changes.

The following methods implement the exclusion list attached to a token.

59b ⟨*Handling exclusion, definition* 59b⟩≡

```
METHOD AddExclusion (Arr: ARRAY OF Token);
METHOD Exclusion: ARRAY OF Token;
```

59c ⟨*Handling exclusion, implementation* 59c⟩≡

```
VAR
  TheExclusion: ARRAY OF Token;

METHOD AddExclusion (Arr: ARRAY OF Token);
  BEGIN
  TheExclusion ← TheExclusion + Arr;
  END AddExclusion;

METHOD Exclusion: ARRAY OF Token;
  BEGIN
  RESULT ← TheExclusion;
  END Exclusion;
```

The following few methods are trivial. They are mere attribute consultation methods or elementary string arithmetics.

59d ⟨*Trivial consultation methods, definition* 59d⟩≡

```
METHOD GetAutomaton: Automaton;
METHOD Image: ARRAY OF CHAR;
METHOD Name: ARRAY OF CHAR;
METHOD Skip: BOOLEAN;
METHOD Error: BOOLEAN;
```

60a ⟨*Trivial consultation methods, implementation* 60a⟩≡

```
METHOD Definition: ARRAY OF CHAR;
  BEGIN
  RESULT ← TheDefinition;
  END Definition;

METHOD GetAutomaton: Automaton;
  BEGIN
  RESULT ← TheAutomaton;
  END GetAutomaton;

METHOD Image: ARRAY OF CHAR;
  BEGIN
  RESULT ← Name + ‘‘ := ” + TheExpr.Image;
  END Image;
```

60b ⟨*Trivial consultation methods, implementation* 60a⟩+≡

```
METHOD Name: ARRAY OF CHAR;
  BEGIN
  RESULT ← Get;
  END Name;

METHOD Skip: BOOLEAN;
  BEGIN
  RESULT ← SkipFlag;
  END Skip;

METHOD Error: BOOLEAN;
  BEGIN
  RESULT ← ErrorFlag;
  END Error;
```

When creating a new token, its number and name must be specified, as well as the `LexSpecification` in which it occurs.

Tokens should never be created directly. They ought to be created by the `ReadStream` method provided by the `LexSpecification` class (See page 63). After that the token has been created, `ReadStream` will also parse the corresponding regular expression by calling the `ParseDefinition` method.

60c ⟨`Token` *creation method definition* 60c⟩≡

```
REDEFINE METHOD CREATE (Nr: INTEGER;
                        Name: ARRAY OF CHAR;
                        Spec: LexSpecification);
```

61a ⟨Token *creation method implementation* 61a⟩≡
```
  REDEFINE METHOD CREATE (Nr: INTEGER;
                          Name: ARRAY OF CHAR;
                          Spec: LexSpecification);
    VAR
      a: ARRAY OF CHAR;
    BEGIN
    ASSERT Name ≠ Λ;
    a ← Name;
    ⟨Consider special token names 61c⟩
    ⟨Call BASE 61b⟩
    TheSpec ← Spec;
    TheNr ← Nr;
    END CREATE;
```

The `BASE` method, that is, the `CREATE` method that is attached to `Token`'s base class, namely, `StringElement` must be called so that `Tokens` can be stored and found within `StringList`'s.

61b ⟨*Call* BASE 61b⟩≡
```
  BASE(a);
```

A leading underscore "_" indicates that the token denotes a space that can separate two other tokens but that should not be returned to the calling program, while a leading exclamation mark "!" indicates that the token should return a standard error code.

61c ⟨*Consider special token names* 61c⟩≡
```
  CASE a0 OF
    ‘‘_’’:
      SkipFlag ← TRUE;
      a ← Name.SLICE (1, Name.SIZE - 1);
      END;
    ‘‘!’’:
      ErrorFlag ← TRUE;
      a ← Name.SLICE (1, Name.SIZE - 1);
      END;
```

In older versions of `lexyt`, a leading dash had a specific meaning which is now obsolete, replaced by more versatile features. We recognize it to produce an ad hoc error message, since quite a few of our existing lexical specifications have not been updated accordingly.

61d ⟨*Consider special token names* 61c⟩+≡
```
    ‘‘-’’:
      StdErr.WriteLine (‘‘Leading dashes in lexical definitions are obsolete’’);
      ErrorFlag ← TRUE;
      END;
   ELSE
    END;
```

At some stage, we considered using generic algorithms such as Earley's [30] that can deal with ambiguous grammars. However, Earley's algorithm is based on the classical model (see figure 2.2 on page 21) with a single lexical interpretation of the input. In order to cope with lexical ambiguities, we must produce Earley parsers where atomic characters serve as lexemes. Such parsers can be obtained by concatenating a `yacc`-like grammar for the syntax analyzer, with a `yacc`-like grammar *for the lexer*. For instance, instead of being based on a lexeme for identifiers, such a grammar will be based on a *non-terminal* for identifiers which definition will be based on atomic characters.

When generating the `yacc` definition corresponding to `THIS`, the best level to work with is at the expression level, with `TheExpr` (See page 55). Calling `OutputYaccRule` (See page 95) will produce something such as:

```
Z4 :  T1 T2 |
      T1 ;
```

We merely produce the token's name, and call `OutputYaccReference` (See page 95). to complete the `yacc` rule by:

```
BEG : Z4 ;
```

if `BEG` is the name of the token for which we are producing the equivalent `yacc` grammar.

62 ⟨`GenerateYacc` *method implementation* 62⟩≡

```
METHOD GenerateYacc (Output: OutputStream);
  BEGIN
  TheExpr.OutputYaccRule (Output);
  Output.WriteString (Get + ‘‘ : ”);
  TheExpr.OutputYaccReference (Output);
  Output.WriteLine (‘‘;”);
  Output.WriteLine (‘‘/*******************/”);
  END GenerateYacc;
```

4.4.2 The `LexSpecification` class

A `LexSpecification` is a container class for tokens by incarnating a list of tokens by inheritance, as well as the variable definitions that can be found within a lexical specification.

After being parsed, such a `LexSpecification` instance can produce successively the corresponding NFA and DFA.

63a ⟨`LexSpecification` *class definition* 63a⟩≡

```
CLASS LexSpecification;
  INHERITS StringList(Token);

  METHOD BuildDfaAutomaton: DfaAutomaton;
  ⟨ReadStream method definition 67c⟩
  ⟨ReadFile method definition 77b⟩

  ⟨FindVariable method definition 63c⟩
  ⟨GenerateYacc method definition for LexSpecification 77d⟩

  REDEFINE METHOD CREATE;

END LexSpecification;
```

63b ⟨`LexSpecification` *class implementation* 63b⟩≡

```
CLASS LexSpecification;
  INHERITS StringList(Token);

  VAR
    VarList: StringList (Variable);

  REDEFINE METHOD CREATE;
    BEGIN
    BASE;
    VarList.CREATE;
    END CREATE;

  ⟨Variable management methods 64a⟩

  ⟨BuildNFAAutomaton method implementation 66a⟩
  ⟨BuildDfaAutomaton method implementation 67b⟩
  ⟨ReadStream method implementation 68a⟩
  ⟨ReadFile method implementation 77c⟩
  ⟨GenerateYacc method implementation for LexSpecification 78a⟩

END LexSpecification;
```

63c ⟨`FindVariable` *method definition* 63c⟩≡

```
METHOD FindVariable (Name: ARRAY OF CHAR): ARRAY OF CHAR;
```

The `FindVariable` method searches for a named variable and should return a string. It first searches through the variables that have been added by using the `AddVariable` method.

Then, if no such variable has been found, the list of tokens is seached. If such a token can be found, its original definition is returned.

In practice, this means that a specification as:

```
$VAR VNR       ([1-9]+)
NR             {VNR}
LABEL          (L|l){VNR}
```

could be written more shortly as:

```
NR             ([1-9]+)
LABEL          (L|l){NR}
```

An existing token can then be reused as is, without having to go through a phony variable (such as `VNR` in the example above).

64a ⟨*Variable management methods* 64a⟩≡

```
  ⟨FindVariable method definition 63c⟩
    VAR
      v: Variable;
      Tok: Token;
    BEGIN
    v ← VarList.FindString (Name);
    IF v = Λ THEN
      Tok ← FindString (Name);
      IF Tok ≠ Λ THEN
        RESULT ← Tok.Definition;
        END;
     ELSE
      RESULT ← v.Value;
      END;
    END FindVariable;
```

The `AddVariable` method takes a variable name and a character string value as parameter, and creates a new variable with that name and value.

It is used for reusable lexical expressions that are not used as tokens by themselves. Hence, they are not returned by the lexical analyzer.

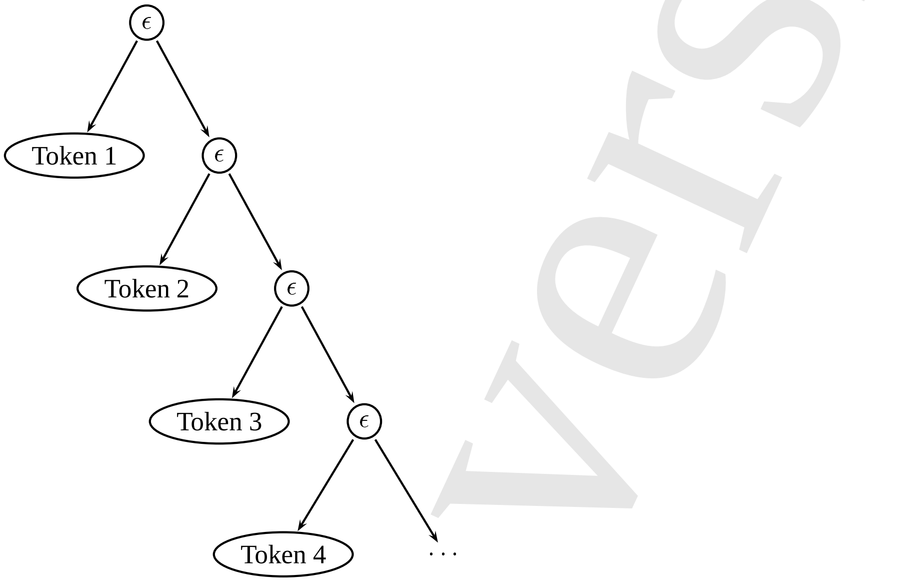
```
$VAR ALPHA      [a-zA-Z]
```

64b ⟨*Variable management methods* 64a⟩+≡

```
  METHOD AddVariable (Key, Value: ARRAY OF CHAR);
    VAR
      v: Variable;
    BEGIN
    v.CREATE (Key, Value);
    VarList.Append (v);
    END AddVariable;
```

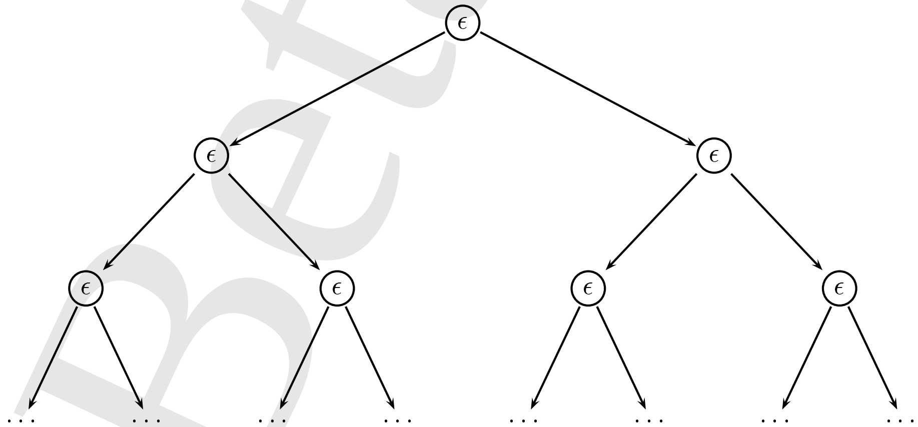
`BuildNFAAutomaton` concatenates all the automata attached to its tokens by building a scaffolding of `EpsNode` instances. However, if this scaffolding is built naively, as shown in figure 4.3, we will experience excessive recursion when trying to find which NFA nodes can be reached before consuming any character from the input.

Figure 4.3 Naive concatenation of the automata of all the tokens of a specification



Rather than unrecursifying the closure algorithm (See page 138), we concentrate in balancing the concatenation, as shown in figure 4.4 so that recursive plunge remains reasonable.

Figure 4.4 Better concatenation of the automata of all the tokens of a specification



66a ⟨`BuildNFAAutomaton` *method implementation* 66a⟩≡
```
METHOD BuildNFAAutomaton: EpsNode;

  ⟨BuildOnSlice local method 66b⟩

  BEGIN
  RESULT ← BuildOnSlice (0, Size-1);
  END BuildNFAAutomaton;
```

`BuildOnSlice` is pretty straightforward: if the slice to convert into an NFA is smaller than 4 (a small arbitrary value), then use the naive scheme as depicted in figure 4.3. Otherwise, we call `BuildOnSlice` recursively, and the two halves are combined trivially.

66b ⟨`BuildOnSlice` *local method* 66b⟩≡
```
METHOD BuildOnSlice (From, To: INTEGER): EpsNode;
  VAR
    Last, ENode1, ENode2: EpsNode;
    Middle: INTEGER;
    Auto: Automaton;
  BEGIN
  ASSERT To > From;
  IF From + 4 ≥ To THEN
    FOR i ∈ From .. To DO
      Auto ← Get(i).GetAutomaton;
      ⟨Make sure that Auto is a decent automaton 66c⟩
      IF RESULT = Λ THEN
        RESULT.CREATE;
        RESULT.Connect (Auto.First);
        Last ← RESULT;
       ELSE
        ENode1.CREATE;
        ENode1.Connect (Auto.First);
        Last.Connect2(ENode1);
        Last ← ENode1;
        END;
      END;
   ELSE
    ⟨Call BuildOnSlice recursively on the two subslices 67a⟩
    END;
  END BuildOnSlice;
```

If `Auto` is a proper automaton, that is, an automaton that has been generated as corresponding to a regular expression, its last node cannot be connected to anything.

66c ⟨*Make sure that* `Auto` *is a decent automaton* 66c⟩≡
```
ASSERT Auto.Last.Connected = Λ;
```

67a ⟨*Call* BuildOnSlice *recursively on the two subslices* 67a⟩≡
```
Middle ← (From + To) / 2;
ENode1 ← BuildOnSlice (From, Middle);
ENode2 ← BuildOnSlice (Middle+1, To);
RESULT.CREATE;
RESULT.Connect (ENode1);
RESULT.Connect2 (ENode2);
```

The deterministic automaton (See class *DfaAutomaton.DfaAutomaton* on page 123) is built after the non-deterministic automaton, returned by the BuildNFAAutomaton method (See page 65). The actual processing is performed by CREATE as redefined by DfaAutomaton (See page 129).

67b ⟨BuildDfaAutomaton *method implementation* 67b⟩≡
```
METHOD BuildDfaAutomaton: DfaAutomaton;
  BEGIN
  RESULT.CREATE (BuildNFAAutomaton, THIS);
  IF YLexCfg.Verbose THEN
    StdOut.WriteLine (‘‘DFA generated”);
    END;
  END BuildDfaAutomaton;
```

Here starts the code that takes an input stream, and parses a LexSpecification from it.

67c ⟨ReadStream *method definition* 67c⟩≡
```
METHOD ReadStream (Input: InputStream);
```

68a ⟨ReadStream *method implementation* 68a⟩≡

```
METHOD ReadStream (Input: InputStream);

  ⟨ReadStream locals 68b⟩
  ⟨Error handling methods 76b⟩

  ⟨Trim method implementation, to remove blanks and tabs 69a⟩
  ⟨NextWord method implementation, to fetch the next word of Line 69b⟩
  ⟨DoClass method implementation, to handle $CLASS lines 73a⟩
  ⟨DoExclusion method implementation, to handle $EXCLUDE lines 76a⟩
  ⟨DollarLine method implementation 71a⟩
  ⟨PatternLine method implementation, to parse a token with its spec 70a⟩

BEGIN
IF YLexCfg.Verbose THEN
  StdOut.WriteLine (‘‘Reading tokens...”);
  END;
WHILE ¬ Input.Eof DO
  TextLine ← Input.ReadLine;
  Line ← Trim(TextLine);
  IF (Line ≠ Λ) ∧ (Line.SIZE > 0) THEN
    CASE Line0 OF
      ‘‘;”, ‘‘#”:   -- Comment, ignore line
        END;
      ‘‘$”:
        DollarLine;
        END;
      ‘‘a” .. ‘‘z”, ‘‘A” .. ‘‘Z”, ‘‘-”, ‘‘_”, ‘‘!”:
        PatternLine;
        END;
     ELSE -- Other characters are not recognized.
      SyntaxError (4);
      END;
    END;
  END;
END ReadStream;
```

TextLine denotes the line that has been read from the file, while Line denotes the part of it that has not been analyzed yet. As elements of the input are recognized, Line is altered while TextLine remains untouched.

TextLine is only maintained so that it can be used to issue reasonably comprehensive error messages. Line numbers would have done the trick as well.

68b ⟨ReadStream *locals* 68b⟩≡

```
VAR
  TextLine, Line: ARRAY OF CHAR;
```

`Trim` is a very simple method, that translates tabs into spaces, and then calls the `Strip` method as defined by the `String` once class to remove leading and trailing blanks.

69a ⟨`Trim` *method implementation, to remove blanks and tabs* 69a⟩≡

```
METHOD Trim (Str: ARRAY OF CHAR): ARRAY OF CHAR;
  BEGIN
  FOR i ∈ 0 .. Str.SIZE - 1 | Str_i = SYSTEM.CHR(SYSTEM.Tab) DO
    Str_i ← ‘‘ ’’;
    END;
  RESULT ← String.Strip (Str);
  END Trim;
```

`NextWord` extracts the front word of `Line`, by checking for valid characters. It returns the front word, and alters `Line` by removing the front word, so that a subsequent call to `NextWord` would return the second word, etc ...

69b ⟨`NextWord` *method implementation, to fetch the next word of* `Line` 69b⟩≡

```
METHOD NextWord: ARRAY OF CHAR;

  ⟨IsLegalCh method, to check for valid characters within an identifier 69c⟩

  VAR
    i: INTEGER;
  BEGIN
  IF Line ≠ Λ THEN
    -- The first character is always assumed valid.
    i ← 1;
    WHILE (i < Line.SIZE) ∧ IsLegalCh(Line_i) DO
      i ← i + 1;
      END;
    IF i ≥ Line.SIZE THEN
      RESULT ← Line;
      Line ← Λ;
     ELSE
      RESULT ← Line.SLICE (0, i);
      Line ← Trim(String.Right(Line, Line.SIZE -i));
      END;
    END;
  END NextWord;
```

69c ⟨`IsLegalCh` *method, to check for valid characters within an identifier* 69c⟩≡

```
METHOD IsLegalCh(Ch: CHAR): BOOLEAN;
  BEGIN
  RESULT ← SYSTEM.IsLetter(Ch) ∨
           SYSTEM.IsDigit(Ch) ∨
           (Ch = ‘‘_’’) ∨ (Ch = ‘‘!’’) ∨ (Ch = ‘‘-’’) ∨
           (Ch = ‘‘$’’);
  END IsLegalCh;
```

`PatternLine` handles the case where the line is not empty and should contain a pattern definition, of the form `Token RegExp` where `Token` denotes the symbolic name that should be given to the token, while `RegExp` denotes its regular expression in human readable form.

70a ⟨`PatternLine` *method implementation, to parse a token with its spec* 70a⟩≡
```
METHOD PatternLine;
   VAR
      Key: ARRAY OF CHAR;
      Tok: Token;
   BEGIN
   Key ← NextWord;
   IF ⟨Check for error in PatternLine 70b⟩ THEN
      SyntaxError (3);
    ELSE
      ⟨Create a new token 70c⟩
      ⟨Set the new token's regular expression 70d⟩
      Append (Tok);
      END;
   END PatternLine;
```

The various conditions that could invalidate a line are:

- `VOID` or empty key
- `VOID` or empty `Line`, that is, no regular expression.

70b ⟨*Check for error in* `PatternLine` 70b⟩≡
```
  (Key = Λ)  ∨ (Key.SIZE = 0)  ∨
    (Line = Λ)  ∨ (Line.SIZE = 0)
```

When a new token is created, its enclosing `LexicalSpecification` instance is given as third parameter (`THIS`).

70c ⟨*Create a new token* 70c⟩≡
```
Tok.CREATE (Size, Key, THIS);
```

After that `NextWord` has been used to fetch the first word on the line, `Line` holds the rest of the line, that is, the token's attached regular expression.

70d ⟨*Set the new token's regular expression* 70d⟩≡
```
Tok.ParseDefinition (Line);
```

The `DollarLine` method handles meta-commands that start with the dollar character "`$`". It is a pretty simplistic method, that extracts the first word of the line using the `NextWord` method, and compares it to the various "`$`"-prefixed commands.

This is of course not very efficient nor scalable, but there are less than a dozen of such meta-commands in a typical lexical specification, so . . .

71a ⟨`DollarLine` *method implementation* 71a⟩≡

```
METHOD DollarLine;
  VAR
    a, Key, Pattern: ARRAY OF CHAR;
    Level: INTEGER;
  BEGIN
  Key ← String.UpperCase(NextWord);
  IF String.Equals (Key, ‘‘$VAR”) THEN
    ⟨Define a variable using the $VAR command 71b⟩
   ELSIF String.Equals (Key, ‘‘$BACKTRACKLEVEL”) THEN
    ⟨Set the backtrack level 72a⟩
   ELSIF String.Equals (Key, ‘‘$IGNORECASE”) THEN
    YLexCfg.SetIgnoreCase (TRUE);
   ELSIF String.Equals (Key, ‘‘$NOIGNORECASE”) THEN
    YLexCfg.SetIgnoreCase (FALSE);
   ELSIF String.Equals (Key, ‘‘$CASE”) THEN
    YLexCfg.SetIgnoreCase (FALSE);
   ⟨Consider $CLASS and $EXCLUDE 72b⟩
   ELSE
    Error (‘‘Unrecognized $ command”);
    END;
  END DollarLine;
```

The `$VAR` command can be used to define variables that can be referred to by other lexical definition by putting between braces.

```
$VAR  Digit   ([0-9])
Integer       {Digit}+
```

71b ⟨*Define a variable using the* `$VAR` *command* 71b⟩≡

```
Key ← NextWord;
```

`Line` then receives the remaining of the line, as a side effect of the invocation of `NextWord` (See page 69)

71c ⟨*Define a variable using the* `$VAR` *command* 71b⟩+≡

```
Pattern ← Line;
IF (Key ≠ Λ) ∧ (Pattern ≠ Λ) THEN
  AddVariable (Key, Pattern);
 ELSE
  SyntaxError(2);
  END;
```

As to the backtrack level, the argument is read as the next word on the line, converted into an integer and checked for validity before being set to `YLexCfg` (See page 175).

72a ⟨*Set the backtrack level* 72a⟩≡

```
a ← NextWord;
IF a ≠ Λ THEN
  IF SYSTEM.UCASE(a_0) = "D" THEN
    YLexCfg.SetDynamicBackTrackLevel (TRUE);
   ELSE
    Level ← IntConversions.StringToInt (a);
    IF (Level ≥ 0) ∧ (Level < 4) THEN
      YLexCfg.SetBackTrackLevel (Level);
     ELSE
      Error ("Invalid backtrack level");
      END;
    END;
  END;
```

The `$CLASS` and `$EXCLUDE` meta-commands are handled by ancillary methods `DoClass` (See the next page) and `DoExclusion` (See page 76).

72b ⟨*Consider* `$CLASS` *and* `$EXCLUDE` 72b⟩≡

```
ELSIF String.Equals (Key, "$CLASS") THEN
  DoClass;
ELSIF String.Equals (Key, "$EXCLUDE") THEN
  DoExclusion;
```

The `DoClass` method handles lines that were marked with the `$CLASS` keyword. It parses them, adds the token to the specification, and builds the associated automaton by calling the `MakeClass` method.

73a ⟨`DoClass` *method implementation, to handle* `$CLASS` *lines* 73a⟩≡

```
METHOD DoClass;
   VAR
     Err: BOOLEAN;
     Key: ARRAY OF CHAR;
     Words,
     Toks: ARRAY OF ARRAY OF CHAR;
     TokList: StringList(Token);
     Tok, OtherTok: Token;
     FirstIndex, LastIndex, TempInt: INTEGER;
   BEGIN
   ⟨Get the new token identifier into Key 73b⟩
   IF (Key = Λ) ∨ (Line = Λ) THEN
     SyntaxError(1);
    ELSE
     ⟨Build the list of words, separated by spaces into Words 73c⟩
     TokList.CREATE;
     FOR Word ∈ Words WHILE ¬ Err DO
        ⟨Check for intervals, set the words in Toks 74a⟩
        IF Toks.SIZE = 1 THEN
           ⟨Handle a single token to add to the class 74b⟩
          ELSIF Toks.SIZE = 2 THEN
           ⟨Handle an interval of tokens to be added to the class 75a⟩
          ELSE   -- Can only be one or two words...
           SyntaxError (6);
           Err ← TRUE;
           END;
        END;
     ⟨Gather TokList, make a token out of it, and append it to THIS 75d⟩
     END;
   END DoClass;
```

The token's name can be fetched from the input line by using the `NextWord` method, that will also truncate `Line` so that it contains the rest of the input line.

73b ⟨*Get the new token identifier into* `Key` 73b⟩≡

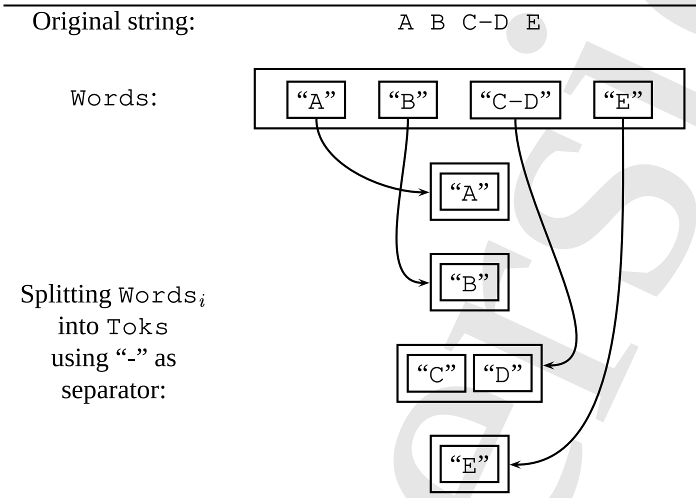
```
Key ← NextWord;
```

After removing the leading `$CLASS` keyword (see above), the words on the line are split twice, once using the space or the comma, once using the dash as delimiter to denote intervals of tokens (See figure 4.5).

73c ⟨*Build the list of words, separated by spaces into* `Words` 73c⟩≡

```
Words ← String.BreakInWords (Line,
                              SkipSeparator ← “ ,”,
                              KeepSeparator ← Λ);
```

Figure 4.5 Using `BreakInWords`



The `BreakInWords` method is used again, to separate the dash-delimited words, and sets the result in `Toks`. If we are dealing with a single word, `Toks` will be an array of one string. If we are dealing with two words separated by a dash, `Toks` will be an array of two elements made of the two words. Any other situation would be an error.

74a ⟨*Check for intervals, set the words in* `Toks` 74a⟩≡

```
Toks ← String.BreakInWords (Word, ''–", Λ);
```

`FindString` searches for a token in a specification by taking advantage of the inheritance relationship between `LexSpecification` and the `StringList` class provided by the YAFL standard library.

74b ⟨*Handle a single token to add to the class* 74b⟩≡

```
Tok ← FindString (Toks₀);
```

If a single token must be added to the class of tokens, the corresponding token is searched in the existing specification. If it can be found, it is appended to `TokList`. Otherwise, an error message is issued.

74c ⟨*Handle a single token to add to the class* 74b⟩+≡

```
IF Tok = Λ THEN
  UndefinedToken (Toks₀);
  Err ← TRUE;
 ELSE
  TokList.Append (Tok);
  END;
```

If the two tokens `Tok` and `OtherTok` have been founds within the specification, their index is searched, compared and swapped if not sorted properly. Then, all the tokens lying between the two bounds are appended to `TokList`.

75a ⟨*Handle an interval of tokens to be added to the class* 75a⟩≡
```
  ⟨Find the tokens in the specification, set Tok and OtherTok 75b⟩
  IF ¬ Err THEN
    FirstIndex ← Index (Tok);
    LastIndex ← Index(OtherTok);
    IF FirstIndex > LastIndex THEN
      ⟨Swap FirstIndex and LastIndex 75c⟩
      END;
    FOR j ∈ FirstIndex .. LastIndex DO
      TokList.Append (THIS_j;
      END;
    END;
```

We first look for the two tokens with the proper name in the specification.

75b ⟨*Find the tokens in the specification, set* `Tok` *and* `OtherTok` 75b⟩≡
```
  Tok ← FindString (Toks_0);
  OtherTok ← FindString (Toks_1);
  IF Tok = Λ THEN
    UndefinedToken (Toks_0);
    Err ← TRUE;
   ELSIF OtherTok = Λ THEN
    UndefinedToken (Toks_1);
    Err ← TRUE;
    END;
```

If the two tokens specified in the range are not sorted, their indexes are swapped.

75c ⟨*Swap* `FirstIndex` *and* `LastIndex` 75c⟩≡
```
  TempInt ← FirstIndex;
  FirstIndex ← LastIndex;
  LastIndex ← TempInt;
```

If no error was encountered while gathering all the tokens of the token class, create a corresponding token by calling the `MakeClass` method. The token can then be added to the `LexicalSpecification` instance denoted by `THIS`, as any other lexical specification.

75d ⟨*Gather* `TokList`*, make a token out of it, and append it to* `THIS` 75d⟩≡
```
  IF ¬ Err THEN
    Tok.CREATE (Size, Key, THIS);
    Tok.MakeClass (TokList.Row);
    Append (Tok);
    END;
```

The `DoExclusion` method handles lines that are marked with the `$EXCLUDE` keyword (See page 35). It parses them, and adds the corresponding exclusion information to the lexical specification.

76a ⟨`DoExclusion` *method implementation, to handle* `$EXCLUDE` *lines* 76a⟩≡

```
METHOD DoExclusion;
  VAR
    a, RootKey: ARRAY OF CHAR;
    RootTok, OtherTok: Token;
    ExcludeList: List(Token);
  BEGIN
  RootKey ← NextWord;
  ExcludeList.CREATE;
  RootTok ← FindString (RootKey);
  IF RootTok = Λ THEN
    UndefinedToken (RootKey);
   ELSE
    a ← NextWord;
    WHILE a ≠ Λ DO
      OtherTok ← FindString (a);
      IF OtherTok = Λ THEN
        UndefinedToken (a);
       ELSE
        ExcludeList.Append (OtherTok);
        END;
      a ← NextWord;
      END;
    END;
  RootTok.AddExclusion (ExcludeList.Row);
  END DoExclusion;
```

The `Error` method displays the offending input line, and error message, and sets a global error flag so that subsequent processing can be cancelled.

76b ⟨*Error handling methods* 76b⟩≡

```
METHOD Error (Msg: ARRAY OF CHAR);
  BEGIN
  StdErr.WriteLn;
  StdErr.WriteLine (TextLine);
  StdErr.WriteLine (Msg);
  YLexCfg.SetError (TRUE);
  END Error;
```

SyntaxError and UndefinedToken are conveniency methods that refer trivially to Error.

77a ⟨*Error handling methods* 76b⟩+≡
```
METHOD SyntaxError (Code: INTEGER);
  BEGIN
  Error (``Syntax error: '' + IntConversions.IntToString (Code, 0));
  END SyntaxError;

METHOD UndefinedToken (Tok: ARRAY OF CHAR);
  BEGIN
  Error (``Undefined token: '' + Tok);
  END UndefinedToken;
```

The ReadFile method is just a conveniency wrapper around the ReadStream method.

77b ⟨ReadFile *method definition* 77b⟩≡
```
METHOD ReadFile (FName: ARRAY OF CHAR);
```

77c ⟨ReadFile *method implementation* 77c⟩≡
```
METHOD ReadFile (FName: ARRAY OF CHAR);
  VAR
    Input: InputStream;
  BEGIN
  Input.CREATE;
  Input.Open (FName, Input.ReadAccess);
  IF Input.ErrorCode > Input.NoError THEN
    StdErr.WriteLine (``Cannot open file: '' + FName);
    YLexCfg.SetError (TRUE);
   ELSE
    ReadStream (Input);
    END;
  END ReadFile;
```

GenerateYacc produces an equivalent yacc source file for a LexSpecification. It basically generates %token commands for the various characters that can occur, and call GenerateYacc (See page 62) for all the tokens in the specification.

77d ⟨GenerateYacc *method definition for* LexSpecification 77d⟩≡
```
METHOD GenerateYacc (FName: ARRAY OF CHAR);
```

78a ⟨GenerateYacc *method implementation for* LexSpecification 78a⟩≡

```
METHOD GenerateYacc (FName: ARRAY OF CHAR);
  VAR
    Output: OutputStream;
  BEGIN
  Output.CREATE;
  Output.Create (FName, Output.WriteAccess);
  FOR i ∈ 0 .. YLexCfg.MaxTokenCode DO
    ⟨Once in a while, start a new line 78b⟩
    Output.WriteString (‘‘ T”);
    Output.WriteInt (i, 0);
    END;
  Output.WriteLn;
  Output.WriteLn;
  Output.WriteLine (‘‘%%”);
  Output.WriteLn;
  FOR t ∈ THIS DO
    t.GenerateYacc (Output);
    END;
  Output.Close;
  END GenerateYacc;
```

78b ⟨*Once in a while, start a new line* 78b⟩≡

```
IF i MOD 6 = 0 THEN
  Output.WriteLn;
  Output.WriteString (‘‘%token”);
  END;
```

4.4.3 The `Variable` private class

The `Variable` class is used within `LexSpecification` to denote variables. The class is private. It is not accessible from the outside world, that can access all variable-related facilities provided by `LexSpecification`.

It is a very simple class that implements a coupling between a variable name and a string value that will be pasted whenever the variable will be used. `Variable` inherits from `StringElement` in order to be able to reuse lookup facilities within `StringList` instances.

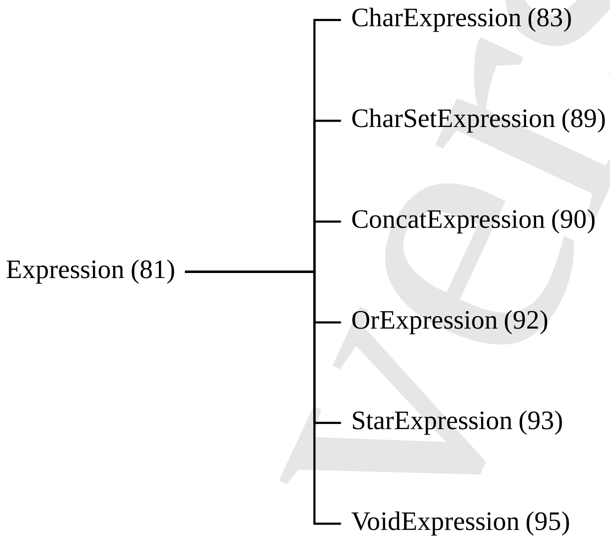
79 ⟨`Variable` *private class implementation* 79⟩≡

```
CLASS Variable;
  INHERITS StringElement;

  VAR
    TheValue: ARRAY OF CHAR;

  REDEFINE METHOD CREATE (Var, Value: ARRAY OF CHAR);
    BEGIN
    BASE (Var);
    TheValue ← Value;
    ASSERT Value ≠ Λ;
    END CREATE;

  METHOD Value: ARRAY OF CHAR;
    BEGIN
    RESULT ← TheValue;
    END Value;

END Variable;
```

4.5 Expression

The `Expression` class describes regular expressions in a tree-like fashion. All the operators commonly found in regular expressions are found as `Expression`-derived classes, as shown in figure 4.6.

Figure 4.6 Inheritance tree for `Expression`



⟨*Definition* ⟩≡
```
DEFINITION MODULE Expression;

FROM Automaton IMPORT Automaton;
IMPORT CharSet;
FROM Streams IMPORT OutputStream;
FROM Token IMPORT LexSpecification;

   ⟨Expression base class definition 81⟩

   ⟨CharExpression class definition 83c⟩
   ⟨CharSetExpression class definition 89b⟩
   ⟨ConcatExpression class definition 90c⟩
   ⟨OrExpression class definition 92a⟩
   ⟨StarExpression class definition 93b⟩
```

Beside these expression classes, the `Expression` module also provides the `ExpressionParser` class that takes a regular expression as a character string and returns the corresponding `Expression`-based tree.

⟨*Definition* ⟩+≡

```
    ⟨ExpressionParser class definition 99a⟩

  END Expression;
```

⟨*Implementation* ⟩≡

```
  IMPLEMENTATION MODULE Expression;

  IMPORT YLexCfg;
  FROM Automaton IMPORT Automaton;
  FROM Node IMPORT Node, CharNode, EpsNode, CharSetNode;
  FROM Conversions IMPORT IntConversions;
  IMPORT Stack;
  FROM Streams IMPORT StdOut, StdErr;
  IMPORT SYSTEM;

    ⟨Expression base class implementation 83a⟩

    ⟨CharExpression class implementation 84a⟩
    ⟨CharSetExpression class implementation 90a⟩
    ⟨ConcatExpression class implementation 91a⟩
    ⟨OrExpression class implementation 92b⟩
    ⟨StarExpression class implementation 93c⟩
    ⟨VoidExpression class implementation 95a⟩
    ⟨ExpressionParser class implementation 99b⟩

  END Expression;
```

4.5.1 The `Expression` base class

The deferred method `BuildAutomaton` builds a NFA equivalent to the expression it is applied to.

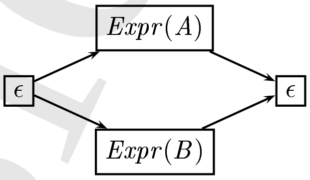
Figure 4.7 shows the treatment of the various regular expression operators when being translated onto an automaton. There is no specific pattern for $A+$: at parse time, it is translated onto $A.A*$. We have a class `StarExpression` (See page 93) to describe $A*$, but we have no such thing as a `PlusExpression` to describe $A+$.

81 ⟨`Expression` *base class definition* 81⟩≡

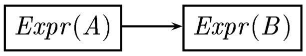
```
  CLASS Expression;
```

Figure 4.7 Regular expressions to NFA

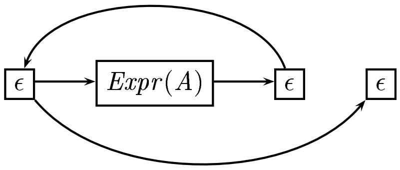
`OrExpression`$(See page\ 92) : Expr(A \mid B) =$



`ConcatExpression`$(See page\ 90) : Expr(A.B) =$



`StarExpression`$(See page\ 93) : Expr(A*) =$



`Expression`'s generic services are:

- Numbering. Every `Expression` has a unique number that can be used to refer to it unambiguously
- Producing a `yacc` identifier for the expression
- Producing the header for a `yacc` grammar rule for the expression
- Mapping to character strings for convenient debugging
- Producing an equivalent automaton

Of all these services, the first three are provided by a universal implementation, while the last two are redefined on a class per class basis.

82a ⟨`Expression` *base class definition* 81⟩+≡
```
  DEFERRED METHOD BuildAutomaton: Automaton;
```

`Image` must be redefined by all derived classes to return a sensible – understand: human readable – text representation of `Expressions`.

82b ⟨`Expression` *base class definition* 81⟩+≡
```
  DEFERRED METHOD Image: ARRAY OF CHAR;
```

82c ⟨`Expression` *base class definition* 81⟩+≡
```
  ⟨yacc generation methods, definition 95b⟩
  END Expression;
```

83a ⟨Expression *base class implementation* 83a⟩≡
```
    CLASS Expression;

      ⟨Numbering facility ... 83b⟩
      ⟨Default yacc generation methods 95c⟩

    END Expression;
```

Uniqueness of expression numbering is achieved by using a once variable `Counter`.

83b ⟨*Numbering facility ...* 83b⟩≡
```
  VAR
    TheNr: INTEGER;

  METHOD Nr: INTEGER;
    VAR
      Counter: ONCE INTEGER;
    BEGIN
    IF TheNr = 0 THEN
      Counter ← Counter + 1;
      TheNr ← Counter;
      END;
    RESULT ← TheNr;
    END Nr;
```

4.5.2 `CharExpression`

`CharExpression` represents the simplest possible expression, made of a character. In order to provide support for extensive character sets beyond ASCII, the actual character is represented by a YAFL `INTEGER` rather than by a `CHAR`.

83c ⟨CharExpression *class definition* 83c⟩≡
```
  CLASS CharExpression;
    INHERITS Expression;
    REDEFINE METHOD CREATE (Code: INTEGER);
  END CharExpression;
```

84a ⟨CharExpression *class implementation* 84a⟩≡
```
CLASS CharExpression;
  INHERITS Expression;

  VAR
    TheCode: INTEGER;

  REDEFINE METHOD CREATE (Code: INTEGER);
    BEGIN
    BASE;
    TheCode ← Code;
    END CREATE;
```

If the word length of the lexical analyzer is 8, then the automaton corresponding to a `CharExpression` is a canonical `CharNode`. We first check for this simple case, before considering cases where the word length is shorter.

84b ⟨CharExpression *class implementation* 84a⟩+≡
```
REDEFINE METHOD BuildAutomaton: Automaton;
  VAR
    First, Last, ChNode: CharNode;
    WLen, TempCode, Div: INTEGER;
    Codes: ARRAY OF INTEGER;
  BEGIN
  WLen ← YLexCfg.WordLength;
  IF WLen = 8 THEN
    ChNode.CREATE (TheCode);
    RESULT.CREATE (ChNode, ChNode);
   ELSE
    ⟨Consider alternate word lengths 88a⟩
    END;
  END BuildAutomaton;
```

By scanning the newsgroup related to compiler technology [44], it appeared to me that quite a few discussions were about handling 16 bits lexers, that is, lexers for which the character is a 16 bits integer, instead of the more common 8 or even 7 bits lexer, as produced by `lex` , for instance.

One of the major issues is efficient and compact representation of the lexer transition function. Given a set of states S, $lextran$ is a partial function:

$$lextran : S \times Alphabet \mapsto S$$

The most straightforward way of implementing it is to use an array of $\#S \times \#Alphabet$ entries. This naive way offers excellent performance, but might require huge data structures.

Since I was myself busy with `lexyt` by then, I had the natural temptation to see how it could be used, perhaps after being slightly adapted, to cope with problems beyond its original scope of application.

My solution was simple: instead of considering 16 bits integers as characters of my alphabet (See page 13), where the transition function $lextran(s, c)$ ($s \in S$, $c \in Alphabet$) has a domain of 2^{16} values for c, it would be preferable to split these 16 bits in two 8 bits-sequences c_1 and c_2 defined as:

$$\begin{array}{|l}
c_1, c_2 : \mathbb{N}_8 \\
\hline
c_1 = high_{16}(c) \\
c_2 = low_{16}(c)
\end{array}$$

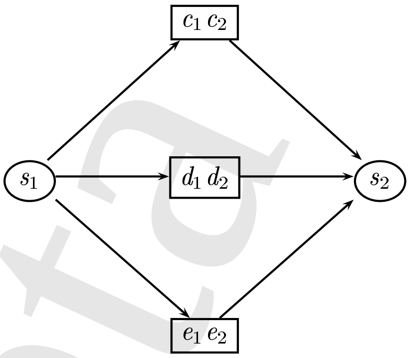
[1]we would then use a function $lextran'(s, c)$ defined as

$$\begin{array}{|l}
lextran' : S \times \mathbb{N}_8 \mapsto S \\
\hline
lextran(s, c) = lextran'(lextran'(s, high_{16}(c)), low_{16}(c))
\end{array}$$

[2]with a domain of 2^8 values for the second argumend.

Of course, such a scheme would yield twice as many transitions (the two evaluations of $lextran'$ for every evaluation of $lextran$) but hopefully, the transition function $lextran'$ could be expressed by a much smaller table than $lextran$, making common table compression techniques as described in and [45] unnecessary, and thereby providing better overall performance.

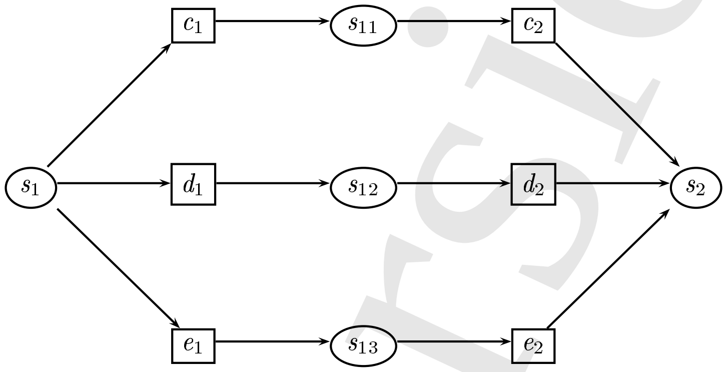
Figure 4.8 16 bits lexer



The rationale behind the hope that the transition table might become smaller was simple. Figure 4.8 depicts a piece of a DFA, with two states between which three 16 bits tokens $c_1\,c_2$, $d_1\,d_2$ and $e_1\,e_2$ are allowed, each of which can be decomposed into two 8 bits words (c_1, c_2, d_1, d_2, e_1 and e_2).

[1]Where we implicitely use the relationship $\#Alphabet = \#\mathbb{N}_{16}$ to consider c as a $\mathbb{N}_{16}$.
[2]$\mathbb{N}_{16}$, low_{16} and $high_{16}$ are defined in page 977.

Figure 4.9 Worst case corresponding 8 bit



In the worst case, depicted in figure 4.9, three additional states s_{11}, s_{12} and s_{13} are created. Extrapolating this example, when translating a 16 bits automaton with $T = \#lextran$ transitions, in the worst case, the size of the translation table goes from $2^{16}.\#S$ to $2^8.(\#S + T)$.

Globally, there will be a gain if $\#S + T < \#S \times 2^8$ or $T < \#S \times (2^8 - 1)$ which is of course, not an easy thing to decide, since, as far as I know, there is no study relating the number of states $\#S$ to the number of transitions T in typical lexical analyzers. Intuitively, it is however clear that an automaton with over 255 transitions in average for every state is quite unlikely.

Besides, things might as well not be as dramatic as in this worst case. Figure 4.10 depicts an example of a more favorable configuration where two of the 16 bits codes have a common 8 bits prefix.

Figure 4.10 More favorable configuration in a 8 bits lexer ($c_1 = d_1$)

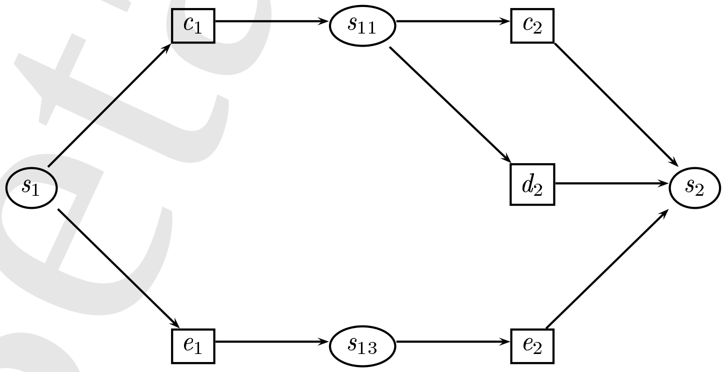


Table 4.1: Sizes of transition tables, when changing the word length

	8 bits	4 bits	2 bits	1 bit
COBOL	463104	71056	34964	$\approx$ 25000
C++	117504	11104	4996	4762
RcScript	46080	4848	2216	2108

This scheme can be extended, to support even shorter word length (4, 2, and ultimately, 1 bit), and table 4.1 gives the table sizes for various lexers, with shorter word lengths[3]. These examples show that the transition tables can be made much smaller than they are, at the expense of a constant factor in performance.

Alternatively [46], a lexer generator might define $lextran'$ as

$$lextran' : S \times \mathbb{N}_8 \mapsto S$$
$$lextran(s, c) = lextran'(lextran'(s, low_{16}(c)), high_{16}(c))$$

instead of

$$lextran' : S \times \mathbb{N}_8 \mapsto S$$
$$lextran(s, c) = lextran'(lextran'(s, high_{16}(c)), low_{16}(c))$$

if it leads to fewer additional states in the automaton. One can generalize this, and have the lexer generator choose the permutation of the sub-slices of the $Alphabet$ that minimizes the number of added transitions.

This is of course not the end of the story about lexers with shorter word lengths. First, an interesting observation. When using 8 bits lexers, by construction, valid lexemes are made of 8 bits characters. When using 4 bits transitions on the same lexical definition, the fact that valid lexemes will always be made of a multiple of 8 bits is guaranteed by the fact that all strings matching one of the so generated patterns will have an even length.

Finally, the most interesting (even if frustrating) result of this entire effort came when I proudly published the results of this study on the internet newsgroup about compilers. Someone – I don't have a copy of this mail any longer, may the friendly anonymous correspondant forgive me, and contact me if he ever reads these lines – kindly explained the `flex` [47] approach to 16 bits lexer generation.

[3] Word length 1 with the `COBOL` lexer did cause `lexyt` to crash on a memory overflow. The figure of 25000 is a rough estimation.

First, the 2^{16} codes are partitioned automatically – that is, by the lexer generator, not by the user – into subsets:

$$\begin{array}{|l}
Part : iseq\,\mathbb{P}\,Alphabet \\
\hline
\forall\, p, q \in \operatorname{ran} Part, p \neq q \bullet p \cap q = \phi \\
\forall\, a \in Alphabet \bullet \exists\, p \in \operatorname{ran} Parts \bullet a \in p \\
\forall\, p \in Part \bullet \\
\qquad \forall\, a, b \in p, \forall\, s \in S \bullet lextran(s, a) = lextran(s, b)
\end{array}$$

The whole idea is to guess that despite the fact that there are 2^{16} different codes, it is highly unlikely that there are as many as 2^{16} different behaviours. Then, the transition function can be implemented by a table of $\#Part \times \#S$ instead of $2^{16} \times \#S$ entries, thereby making more time consuming table compression techniques totally useless.

You understand, beloved reader, my frustration when understanding that a simpler scheme, designed years ago, provided better results than my own ideas. Well, I guess this is the rule of the game when one confronts indigeneous ideas to the external world.

All in all, shorter word lengths are implemented here for measurement purposes only, as in table 4.1. No production lexer has ever used them so far.

After this long digression, let's now see how shorter character lengths are implemented in practice. First, one builds an array made of the character code of the `CharNode` expression, split according to the proper number of bits.

To do so, we use the fact that `TheCode` is positive, and far smaller than the limits of integers. Hence, shifting can be safely performed by division and multiplication.

88a ⟨*Consider alternate word lengths* 88a⟩≡

```
Div ← SYSTEM.SetBit (0, WLen);
Codes.CREATE (8 / WLen);
TempCode ← TheCode;
FOR i ∈ 0 .. Codes.SIZE - 1 DO
  Codes_i ← TempCode MOD Div;
  TempCode ← TempCode / Div;
  END;
ASSERT TempCode = 0;
```

Now, the only thing we must do is to build a sequence of `CharNodes` and to link them together.

88b ⟨*Consider alternate word lengths* 88a⟩+≡

```
FOR i ∈ Codes.SIZE - 1 .. 0 BY -1 DO
  ChNode.CREATE (Codes_i);
  IF First = Λ THEN
    First ← ChNode;
   ELSE
    Last.Connect (ChNode);
    END;
  Last ← ChNode;
  END;
RESULT.CREATE (First, Last);
```

Image, as redefined by CharExpression returns a readable representation of the character transition, as the character itself between parentheses if it is printable, and its ASCII code preceded by a backslash otherwise.

89a ⟨CharExpression *class implementation* 84a⟩+≡

```
REDEFINE METHOD Image: ARRAY OF CHAR;
  BEGIN
  IF (TheCode ≤ 32) ∨ (TheCode > 127) THEN
    RESULT ← ‘‘(\” + IntConversions.IntToString(TheCode, 0)
                   + ‘‘)”;
   ELSE
    RESULT.CREATE(3);
    RESULT_0 ← ‘‘(”;
    RESULT_1 ← SYSTEM.CHR(TheCode);
    RESULT_2 ← ‘‘)”;
    END;
  END Image;

  ⟨yacc generation for CharExpression 96b⟩

END CharExpression;
```

4.5.3 CharSetExpression

CharSetExpression denotes an expression made of a character set, actually represented by the CharSet class (See page 120).

89b ⟨CharSetExpression *class definition* 89b⟩≡

```
CLASS CharSetExpression;
  INHERITS Expression;
  REDEFINE METHOD CREATE (TheCharSet: CharSet);
END CharSetExpression;
```

90a ⟨CharSetExpression *class implementation* 90a⟩≡

```
CLASS CharSetExpression;
  INHERITS Expression;

  VAR
    TheCharSet: CharSet;

  REDEFINE METHOD CREATE (TheCharSet: CharSet);
    BEGIN
    BASE;
    THIS.TheCharSet ← TheCharSet;
    END CREATE;

  REDEFINE METHOD Image: ARRAY OF CHAR;
    BEGIN
    RESULT ← ''{Set}'';
    END Image;

  ⟨BuildAutomaton as redefined by CharSetExpression 90b⟩
  ⟨yacc generation for CharSetExpression 97a⟩

END CharSetExpression;
```

The NFA attached to a `CharSetExpression` is simply made of a `CharSetNode` instance (See page 146), where all the intelligence about the actual behaviour of character classes in automata resides.

90b ⟨BuildAutomaton *as redefined by* CharSetExpression 90b⟩≡

```
REDEFINE METHOD BuildAutomaton: Automaton;
  VAR
    ChNode: CharSetNode;
  BEGIN
  ChNode.CREATE (TheCharSet);
  RESULT.CREATE (ChNode, ChNode);
  END BuildAutomaton;
```

4.5.4 ConcatExpression

90c ⟨ConcatExpression *class definition* 90c⟩≡

```
CLASS ConcatExpression;
  INHERITS Expression;

  REDEFINE METHOD CREATE (One, Two: Expression);
END ConcatExpression;
```

The `ConcatExpression` class represents the most common operator among regular expressions, which is concatenation. This operator is implicit in the original lexical specification source code, but must be made explicit in the corresponding intermediate representation.

91a ⟨`ConcatExpression` *class implementation* 91a⟩≡

```
CLASS ConcatExpression;
  INHERITS Expression;

  VAR
    One, Two: Expression;

  REDEFINE METHOD CREATE (One, Two: Expression);
    BEGIN
    BASE;
    ASSERT One ≠ Λ;
    ASSERT Two ≠ Λ;
    THIS.One ← One;
    THIS.Two ← Two;
    END CREATE;
```

The `BuildAutomaton` method, as redefined by the `ConcatExpression` class is quite straightforward: it simply concatenates the two automatas of `One` and `Two` (see figure 4.7 on page 82).

91b ⟨`ConcatExpression` *class implementation* 91a⟩+≡

```
  REDEFINE METHOD BuildAutomaton: Automaton;
    VAR
      Left, Right: Automaton;
    BEGIN
    Left ← One.BuildAutomaton;
    Right ← Two.BuildAutomaton;
    Left.Last.Connect(Right.First);
    RESULT.CREATE (Left.First, Right.Last);
    END BuildAutomaton;

  REDEFINE METHOD Image: ARRAY OF CHAR;
    BEGIN
    RESULT ← One.Image + ‘‘+’’ + Two.Image;
    END Image;

  ⟨yacc generation for ConcatExpression 97b⟩

END ConcatExpression;
```

4.5.5 OrExpression

The `OrExpression` class implements the alternative $a \mid b$. It is entirely trivial, except for the way it builds the corresponding automaton.

92a ⟨`OrExpression` *class definition* 92a⟩≡

```
CLASS OrExpression;
  INHERITS Expression;

  REDEFINE METHOD CREATE (One, Two: Expression);
END OrExpression;
```

92b ⟨`OrExpression` *class implementation* 92b⟩≡

```
CLASS OrExpression;
  INHERITS Expression;

  VAR
    One, Two: Expression;

  REDEFINE METHOD CREATE (One, Two: Expression);
    BEGIN
    BASE;
    ASSERT One ≠ Λ;
    ASSERT Two ≠ Λ;
    THIS.One ← One;
    THIS.Two ← Two;
    END CREATE;
```

`OrExpression` builds the corresponding automaton according to figure 4.7. First, the automata for its two arguments are created. Then, a leading and a trailing `EpsNode` (See page 143) are created to join resp. the entry and exit points of the automata of the arguments.

92c ⟨`OrExpression` *class implementation* 92b⟩+≡

```
REDEFINE METHOD BuildAutomaton: Automaton;
  VAR
    Start, End: EpsNode;
    Auto1, Auto2: Automaton;
  BEGIN
  Start.CREATE;
  End.CREATE;
  Auto1 ← One.BuildAutomaton;
  Auto2 ← Two.BuildAutomaton;
  Start.Connect (Auto1.First);
  Start.Connect2 (Auto2.First);
  Auto1.Last.Connect (End);
  Auto2.Last.Connect (End);
  RESULT.CREATE (Start, End);
  END BuildAutomaton;
```

93a ⟨OrExpression *class implementation* 92b⟩+≡

```
REDEFINE METHOD Image: ARRAY OF CHAR;
  BEGIN
  RESULT ← ‘‘(” + One.Image + ‘‘|” + Two.Image + ‘‘)”;
  END Image;

  ⟨yacc generation for OrExpression 98a⟩

END OrExpression;
```

4.5.6 StarExpression

The StarExpression class implements the repetition $a*$.

93b ⟨StarExpression *class definition* 93b⟩≡

```
CLASS StarExpression;
  INHERITS Expression;

  REDEFINE METHOD CREATE (One: Expression);
END StarExpression;
```

93c ⟨StarExpression *class implementation* 93c⟩≡

```
CLASS StarExpression;
  INHERITS Expression;

  VAR
    One: Expression;

  REDEFINE METHOD CREATE (One: Expression);
    BEGIN
    BASE;
    ASSERT One ≠ Λ;
    THIS.One ← One;
    END CREATE;
```

Well, just in case you wondered, the construction described here can be made much easier to understand by checking figure 4.7 on page 82.

94 ⟨StarExpression *class implementation* 93c⟩+≡

```
REDEFINE METHOD BuildAutomaton: Automaton;
  VAR
    Start, End1, End2: EpsNode;
    Automat: Automaton;
  BEGIN
  Start.CREATE;
  End1.CREATE;
  End2.CREATE;
  RESULT.CREATE (Start, End2);
  Automat ← One.BuildAutomaton;
  Start.Connect (Automat.First);
  Start.Connect2 (End2);
  Automat.Last.Connect (End1);
  End1.Connect (Start);
  End1.Connect2 (End2);
  RESULT.CREATE (Start, End2);
  END BuildAutomaton;

REDEFINE METHOD Image: ARRAY OF CHAR;
  BEGIN
  RESULT ← ‘‘STAR(” + One.Image + ‘‘)”;
  END Image;

⟨yacc generation for StarExpression 98b⟩

END StarExpression;
```

4.5.7 VoidExpression

VoidExpression is the simplest possible regular expression: it matches the "@" in the input, and its automaton is just made of a single EpsNode (See page 143).

95a ⟨VoidExpression *class implementation* 95a⟩≡

```
CLASS VoidExpression;
  INHERITS Expression;

  REDEFINE METHOD BuildAutomaton: Automaton;
    VAR
      ENode: EpsNode;
    BEGIN
    ENode.CREATE;
    RESULT.CREATE (ENode, ENode);
    END BuildAutomaton;

  REDEFINE METHOD Image: ARRAY OF CHAR;
    BEGIN
    RESULT ← ‘‘(@)”;
    END Image;

  ⟨yacc generation for VoidExpression 98c⟩

END VoidExpression;
```

4.5.8 yacc generation

This section generates equivalent yacc rules for regular expressions (See page 62 for the rationale behind this apparently bizarre idea).

The two following methods implement the yacc generation system.

95b ⟨yacc *generation methods, definition* 95b⟩≡

```
METHOD OutputYaccReference (Output: OutputStream);
METHOD OutputYaccRule (Output: OutputStream);
```

OutputYaccReference outputs the name of the yacc rule attached to the Expression denoted by THIS. It uses the unique numbering scheme (See method *Expression.Expression.Nr* on page 83) to produce unique identifiers.

95c ⟨*Default* yacc *generation methods* 95c⟩≡

```
METHOD OutputYaccReference (Output: OutputStream);
  BEGIN
  Output.WriteChar (‘‘Z”);
  Output.WriteInt (Nr, 0);
  Output.WriteChar (‘‘ ”);
  END OutputYaccReference;
```

OutputYaccRule produces the yacc definition of the rule that describes THIS. The default version of this method merely outputs the rule header, and will be called by redefined versions.

96a ⟨*Default* yacc *generation methods* 95c⟩+≡

```
METHOD OutputYaccRule (Output: OutputStream);
  BEGIN
  OutputYaccReference (Output);
  Output.WriteLine (":");
  END OutputYaccRule;
```

A reference to a CharExpression is replaced by the code for the character, rather than a reference to an external rule. Instead of generating:

```
Z76 : T1;

Z77 : Z02 Z76 Z02 |
      Z76
```

[4]lexyt will generate

```
Z77 : Z02 T1 Z02 |
      T1
```

which is of course strictly equivalent.

96b ⟨yacc *generation for* CharExpression 96b⟩≡

```
REDEFINE METHOD OutputYaccReference (Output: OutputStream);
  BEGIN
  Output.WriteChar ("T");
  Output.WriteInt (TheCode, 0);
  Output.WriteChar (" ");
  END OutputYaccReference;
```

Since all references to a CharExpression are replaced by a reference to the corresponding yacc terminal inline, OutputYaccRule serves no purpose.

96c ⟨yacc *generation for* CharExpression 96b⟩+≡

```
REDEFINE METHOD OutputYaccRule (Output: OutputStream);
  BEGIN
  -- Do nothing...
  END OutputYaccRule;
```

[4]where T1 refers to a terminal

The yacc rule for a character set expression is a disjunction of all the characters in the set.

97a ⟨yacc *generation for* CharSetExpression 97a⟩≡
```
REDEFINE METHOD OutputYaccRule (Output: OutputStream);
  VAR
    First: BOOLEAN;
  BEGIN
  BASE(Output);
  First ← TRUE;
  FOR i ∈ 0 .. YLexCfg.MaxTokenCode - 1 |
        TheCharSet.Includes(i) DO
    IF ¬ First THEN
      Output.WriteLine ("   |");
      END;
    First ← FALSE;
    Output.WriteChar ("T");
    Output.WriteInt (i, 0);
    END;
  Output.WriteLine (";");
  Output.WriteLn;
  END OutputYaccRule;
```

Concatenation is rather trivial . . .

97b ⟨yacc *generation for* ConcatExpression 97b⟩≡
```
REDEFINE METHOD OutputYaccRule (Output: OutputStream);
  BEGIN
  One.OutputYaccRule (Output);
  Two.OutputYaccRule (Output);
  BASE(Output);
  One.OutputYaccReference (Output);
  Two.OutputYaccReference (Output);
  Output.WriteLine (";");
  Output.WriteLn;
  END OutputYaccRule;
```

and so is disjunction.

98a ⟨yacc *generation for* OrExpression 98a⟩≡
```
REDEFINE METHOD OutputYaccRule (Output: OutputStream);
  BEGIN
  One.OutputYaccRule (Output);
  Two.OutputYaccRule (Output);
  BASE (Output);
  One.OutputYaccReference (Output);
  Output.WriteLine ("  |");
  Two.OutputYaccReference (Output);
  Output.WriteLine (";");
  Output.WriteLn;
  END OutputYaccRule;
```

When dealing with a `StarExpression`, the grammar rule for its argument is first generated. Then, if the reference to this argument is `Z71`, a recursive grammar rule of the form:

```
Z72 : Z72 Z71 |
      ;
```

is generated.

98b ⟨yacc *generation for* StarExpression 98b⟩≡
```
REDEFINE METHOD OutputYaccRule (Output: OutputStream);
  BEGIN
  One.OutputYaccRule (Output);
  BASE(Output);
  OutputYaccReference (Output);
  One.OutputYaccRule (Output);
  Output.WriteLine ("  |");
  Output.WriteLine (";");
  Output.WriteLn;
  END OutputYaccRule;
```

98c ⟨yacc *generation for* VoidExpression 98c⟩≡
```
REDEFINE METHOD OutputYaccReference (Output: OutputStream);
  BEGIN
  END OutputYaccReference;

REDEFINE METHOD OutputYaccRule (Output: OutputStream);
  BEGIN
  END OutputYaccRule;
```

4.5.9 ExpressionParser

99a ⟨ExpressionParser *class definition* 99a⟩≡
```
ONCE CLASS ExpressionParser;

  ⟨Parse method definition for ExpressionParser 116c⟩
  ⟨MakeTokenClass method definition 117d⟩
END ExpressionParser;
```

ExprError indicates whether an error has been encountered while parsing the current expression.

99b ⟨ExpressionParser *class implementation* 99b⟩≡
```
ONCE CLASS ExpressionParser;

VAR
  ExprError: BOOLEAN;
```

99c ⟨ExpressionParser *class implementation* 99b⟩+≡
```
⟨Codify method implementation 100a⟩
```

The Codify method takes a character string as parameter, and returns an array of integers that can be used to generate a tree-like expression that describes the same regular expression. In this array, special codes listed below are used to represent operators while simple characters are represented by their ordinal code.

99d ⟨ExpressionParser *class implementation* 99b⟩+≡
```
CONST
  LParen = 10000;     -- (
  RParen = 10001;     -- )
  LBracket = 10002;   -- [
  RBracket = 10003;   -- ]
  Star = 10004;       -- *
  Not = 10005;        -- ^
  Plus = 10006;       -- +
  Or = 10007;         -- |
  Range = 10008;      -- -
  Void =  10009;      -- @
  Any = 10010;        -- .
  NewLine = 10;
  Tab = 9;
```

The purposes of these constants is to represent the input in a codified way. For instance, using the Codify method described below, an input such as (a|b)c can be represented as an array of integers $\boxed{10000}\boxed{97}\boxed{10007}\boxed{98}\boxed{10001}\boxed{99}$, knowing that 97, 98 and 99 are the ASCII codes for resp. “a”, “b” and “c”.

At this stage, the processing is neutral, in the sense there is no interpretation, no parentheses or bracket matching. The only non-trivial task that is performed here is related to variable expansion.

100a ⟨`Codify` *method implementation* 100a⟩≡

```
METHOD Codify(Source: ARRAY OF CHAR;
              Spec: LexSpecification): ARRAY OF INTEGER;

  ⟨Codify locals 100b⟩
  ⟨ExtractWord local method 103b⟩
  ⟨Codify main body 100c⟩
```

`TargetPos` indicates the position where the next integer token will be stored within `RESULT`. Its default initialization of 0 is correct.

`SourcePos` indicates the position within the source string `Source`.

`Idx` and `Len` are local counters.

100b ⟨`Codify` *locals* 100b⟩≡

```
VAR
  TargetPos, SourcePos, Len: INTEGER;
```

Scanning the `Source` array of character is not performed by a FOR loop because the index `SourcePos` must be changed by side effects from within the `ExtractWord` local method. YAFL's FOR loop is much stricter than C's from that point of view. The index variable is read-only from within the loop.

100c ⟨`Codify` *main body* 100c⟩≡

```
BEGIN
⟨Init the RESULT array 101c⟩
WHILE SourcePos < Source.SIZE DO
  CASE Source[SourcePos] OF
    "\":
      ⟨Handle the '\' case 102b⟩
      END;
    "(":
      RESULT[TargetPos] ← LParen;
      END;
    ")":
      RESULT[TargetPos] ← RParen;
      END;
    "[":
      RESULT[TargetPos] ← LBracket;
      END;
    "]":
      RESULT[TargetPos] ← RBracket;
      END;
    "*":
      RESULT[TargetPos] ← Star;
      END;
```

Continued...

101a ⟨`Codify` *main body* 100c⟩+≡
```
  ‘‘^’’:
    RESULT_TargetPos ← Not;
    END;
  ‘‘+’’:
    RESULT_TargetPos ← Plus;
    END;
  ‘‘|’’:
    RESULT_TargetPos ← Or;
    END;
  ‘‘-’’:
    RESULT_TargetPos ← Range;
    END;
  ‘‘@’’:
    RESULT_TargetPos ← Void;
    END;
  ‘‘.’’:
    RESULT_TargetPos ← Any;
    END;
```

When encountering `{`, `lexyt` pastes the content of another variable (See method *Expression.ExpressionParser.ExtractWord* on page 103).

101b ⟨`Codify` *main body* 100c⟩+≡
```
        ‘‘{’’:
          ExtractWord(RESULT);
          END;
      ELSE
        RESULT_TargetPos ← SYSTEM.ORD(Source_SourcePos);
        END;
    SourcePos ← SourcePos + 1;
    TargetPos ← TargetPos + 1;
    END;
  ⟨Truncate the RESULT array 102a⟩
  END Codify;
```

The `RESULT` array is allocated brutally with a size considered as more than large enough (namely, 1024), and initialized with a dummy negative value. To exceed this value, one would need a lexical definition which would contain more than 1024 items after expansion of all macros.

101c ⟨*Init the* `RESULT` *array* 101c⟩≡
```
  RESULT.CREATE (1024);
  FOR Idx ∈ 0 .. RESULT.SIZE - 1 DO
    RESULT_Idx ← -1;
    END;
```

Then, before the array is returned, a minimal slice with no dummy value is computed

102a ⟨*Truncate the* RESULT *array* 102a⟩≡
```
TargetPos ← RESULT.SIZE;
WHILE (TargetPos > 0) ∧ (RESULT[TargetPos-1] = -1) DO
  TargetPos ← TargetPos-1;
  END;
RESULT ← RESULT.SLICE (0, TargetPos);
```

Three special cases are handled when a backslash is used: the c-like convention for the newline, the tab, and an ordinal code. Otherwise, it serves as escape for character that would otherwise be interpreted differently in a regular expression context: '*', '+', '-', etc...

102b ⟨*Handle the '\' case* 102b⟩≡
```
SourcePos ← SourcePos + 1;
IF SourcePos < Source.SIZE THEN
  CASE Source[SourcePos] OF
    "n":
      RESULT[TargetPos] ← NewLine;
      END;
    "t":
      RESULT[TargetPos] ← Tab;
      END;
    "0" .. "9":
      ⟨Handle the '\digit' case 102c⟩
      END;
   ELSE
    RESULT [TargetPos] ← SYSTEM.ORD(Source[SourcePos]);
    END;
  END;
```

Unlike C, the digits that follow a backslash are always considered as decimal digits. The processing stops when the current character is no longer a digit or when the maximal length of 5 (prepared for the support of lexers with transitions on characters up to 16 bits, that is, 65536) has been reached.

102c ⟨*Handle the '\digit' case* 102c⟩≡
```
Len ← 1;
WHILE (SourcePos+Len < Source.SIZE) ∧
      SYSTEM.IsDigit(Source[SourcePos+Len]) ∧
      (Len < 5) DO
  Len ← Len + 1;
  END;
RESULT[TargetPos]← IntConversions.
                      StringToInt(Source.SLICE(SourcePos, Len));
⟨Check for valid ASCII code 103a⟩
SourcePos ← SourcePos + Len - 1;
```

103a ⟨*Check for valid ASCII code* 103a⟩≡

```
ASSERT RESULT_TargetPos ≥ 0;
IF RESULT _TargetPos ≥ YLexCfg.MaxTokenCode THEN
  StdErr.WriteLine (“Illegal ordinal code in ” + Source);
  YLexCfg.SetError (TRUE);
  END;
```

The `ExtractWord` method is called when an opening brace is found. If first finds the matching closing brace, extracts the identifier name between the two braces, and calls the `FindVariable` (See page 63) method.

If a proper variable has been found, it is pasted into the `Res` array.

103b ⟨`ExtractWord` *local method* 103b⟩≡

```
METHOD ExtractWord(Res: ARRAY OF INTEGER);
  VAR
    i: INTEGER;
    a, b: ARRAY OF CHAR;
  BEGIN
  i ← SourcePos + 1;
  ⟨Find matching brace by incrementing i 103c⟩
  IF i < Source.SIZE THEN
    a ← Source.SLICE (SourcePos + 1, i - SourcePos - 1);
    SourcePos ← i;
    b ← Spec.FindVariable (a);
    IF b = Λ THEN
      StdErr.WriteLine (“Undefined variable ” + a);
      YLexCfg.SetError (TRUE);
     ELSE
      ⟨Paste the string b into Res 104a⟩
      END;
   ELSE
    StdErr.WriteLine (“Missing } in ” + Source);
    YLexCfg.SetError (TRUE);
    END;
  END ExtractWord;
```

Braces are used to specify variable pasting. Unlike parentheses, there is no need to consider possible nesting for them.

103c ⟨*Find matching brace by incrementing* `i` 103c⟩≡

```
WHILE (i < Source.SIZE) ∧ (Source_i ≠ “}”) DO
  i ← i + 1;
  END;
```

104a ⟨*Paste the string* `b` *into* `Res` 104a⟩≡

```
FOR v ∈ Codify(b, Spec) DO
  Res_TargetPos ← v;
  TargetPos ← TargetPos + 1;
  END;
TargetPos ← TargetPos - 1;
```

104b ⟨`ExpressionParser` *class implementation* 99b⟩+≡

```
⟨ParseCode method implementation 105a⟩
```

`ParseCode` takes a codified[5] array as parameter, builds the `Expressions` for each of its components, assembles these `Expressions` into a single `Expression` according to implicit concatenation; and returns it.

105a ⟨`ParseCode` *method implementation* 105a⟩≡

```
METHOD ParseCode (Code: ARRAY OF INTEGER): Expression;

⟨ParseCode locals 105b⟩

⟨Stack operations 107c⟩

⟨SearchRParen local method 106⟩
⟨SearchRBracket local method 107a⟩
⟨HandleBracket local method 113b⟩

  BEGIN
  Eval.CREATE;   -- Create the evaluation stack
  WHILE (FromPos < Code.SIZE) ∧ ¬ ExprError DO
    CASE Code[FromPos] OF
      ⟨Simple operators 111c⟩
      LParen:
        ⟨Handling opening parentheses 112b⟩
        END;
      LBracket:
        ⟨Handling opening bracket 113a⟩
        END;
      ⟨Handling error tokens 116a⟩
     ELSE -- Default case
      ASSERT Code[FromPos] ≥ 0;
      ASSERT Code[FromPos] < YLexCfg.MaxTokenCode;
      CharTop(Code[FromPos]);
      END;
    FromPos ← FromPos + 1;
    END;
  IF ¬ ExprError THEN
    ⟨Implicit concatenation of the evaluation stack 111b⟩
    RESULT ← Eval.Pop;
    END;
  END ParseCode;
```

105b ⟨`ParseCode` *locals* 105b⟩≡

```
VAR
  Expr: Expression;
  FromPos, ToPos: INTEGER;
```

[5]That is, an array that has been returned by the `Codify` method (See page 100).

The `SearchRParen` method is a simple utility method that, given an array of integers denoted by `Code` and a start position `InitPos`, assuming that the position indicates an open parenthese, returns the matching closing parenthese's position, taking nesting into account.

If no matching parenthese has been found, `SearchRParen` returns `Code.SIZE` which is an invalid index within `Code`, since YAFL arrays are numbered from `0` to `SIZE-1`.

Unlike many other methods in `ParseCode`, this method is free from any side effect. It does not alter the current pointer within the buffer.

There is no need to consider escaped parentheses "`\(`" and "`\)`" since this is taken care of in a previous phase by the `Codify` method (See page 100), which will use a different integer value whether the character is a plain parenthese or an escaped parenthese.

106 ⟨`SearchRParen` *local method* 106⟩≡

```
METHOD SearchRParen (Code: ARRAY OF INTEGER;
                     InitPos: INTEGER): INTEGER;
  VAR
    i, Level: INTEGER;
  BEGIN
  Level ← 1;
  i ← InitPos + 1;
  WHILE (i < Code.SIZE) ∧ (Level > 0) DO
    IF Code_i = LParen THEN
      Level ← Level + 1;
     ELSIF Code_i = RParen THEN
      Level ← Level - 1;
      RESULT ← i;
      END;
    i ← i + 1;
    END;
  IF Level > 0 THEN
    RESULT ← Code.SIZE;
    END;
  END SearchRParen;
```

The `SearchRBracket` method is even simpler, since it does not have to take nesting into account. It just makes a simple and stupid error detection based on two successive opening brackets (`LBracket`) without a matching closing bracket (`RBracket`)

107a ⟨`SearchRBracket` *local method* 107a⟩≡

```
METHOD SearchRBracket (Code: ARRAY OF INTEGER;
                       InitPos: INTEGER): INTEGER;
  BEGIN
  RESULT ← InitPos + 1;
  WHILE (RESULT < Code.SIZE) ∧ (Code_RESULT ≠ RBracket)
        ∧ (Code_RESULT ≠ LBracket) DO
    RESULT ← RESULT + 1;
    END;
  IF (RESULT < Code.SIZE) ∧ (Code_RESULT = LBracket) THEN
    StdErr.WriteString (“Illegal nested brackets in expression: ”);
    ExprError ← TRUE;
    END;
  END SearchRBracket;
```

`Eval` is an evaluation stack, where intermediate expressions are stored before being combined into more complex expressions, using the various operators.

107b ⟨`ParseCode` *locals* 105b⟩+≡

```
VAR
  Eval: Stack(Expression);
```

The following methods take their implicit arguments from the `Eval` stack described above, and leave their result on that same stack.

`StarTop` pops an expression A, and pushes $A*$ (See class *Expression.StarExpression* on page 93).

107c ⟨*Stack operations* 107c⟩≡

```
METHOD StarTop;
  VAR
    StarExpr: StarExpression;
  BEGIN
  IF Eval.Depth > 0 THEN
    StarExpr.CREATE (Eval.Pop);
    Eval.Push (StarExpr);
   ELSE
    StdErr.WriteString (“No argument for Star operator in expression: ”);
    ExprError ← TRUE;
    END;
  END StarTop;
```

`OrTop` pops two expressions A and B, and pushes $A \mid B$ (See class *Expression. OrExpression* on page 92). Parentheses are taken care separately, that is, if A is a complex expression between parentheses, what remains on the evaluation stack at this stage is the resulting expression of A's full evaluation.

108a ⟨*Stack operations* 107c⟩+≡

```
METHOD OrTop;
   VAR
     OrExpr: OrExpression;
   BEGIN
   IF Eval.Depth > 1 THEN
     ⟨Build the OrExpression instance 108b⟩
     Eval.Push (OrExpr);
    ELSE
     StdErr.WriteString (``Missing argument(s) for " +
                         ``OR operator in expression: ");
     ExprError ← TRUE;
     END;
   END OrTop;
```

The order of evaluation of YAFL actual parameters is not defined by the language definition. However, in this particular case, since the disjunction "|" is commutative, `Pop` can be called twice without the risk of having different behaviors due to the order in which a given YAFL (or `C`) compiler evaluates actuals.

In other circumstances, one might have preferred using two intermediate variables, popping the arguments in a well-defined order, and building the combined expression accordingly.

108b ⟨*Build the* `OrExpression` *instance* 108b⟩≡

```
OrExpr.CREATE (Eval.Pop, Eval.Pop);
```

`PlusTop` pops an expression A, and pushes $A+$.

108c ⟨*Stack operations* 107c⟩+≡

```
METHOD PlusTop;
   BEGIN
   IF Eval.Depth > 0 THEN
     Eval.Dup;
     StarTop;
     ConcatTop;
    ELSE
     StdErr.WriteString (``No argument for + " +
                         ``operator in expression: ");
     ExprError ← TRUE;
     END;
   END PlusTop;
```

`ConcatTop` pops two expressions A and B, and pushes $A.B$ (See class *Expression.ConcatExpression* on page 90). In this case, it is wiser to extract the two arguments required to `CREATE` the `ConcatExpression` instance into temporary variables, thereby ensuring an order in evaluation, since $A.B \neq B.A$ and YAFL does not guarantee the order of evaluation of method actuals.

109a ⟨*Stack operations* 107c⟩+≡
```
METHOD ConcatTop;
  VAR
    Top, AnteTop: Expression;
    ConcatExpr: ConcatExpression;
  BEGIN
  ASSERT Eval.Depth > 1;
  Top ← Eval.Pop;
  AnteTop ← Eval.Pop;
  ConcatExpr.CREATE (AnteTop, Top);
  Eval.Push (ConcatExpr);
  END ConcatTop;
```

The `CharTop` method pushes a single character expression onto the stack (See class *Expression.CharExpression* on page 83). If the flag for the generation of a case-independant lexer is set (See method *YLexCfg.YLexCfg.IgnoreCase* on page 177), it checks whether the character is a letter by testing whether it is different from its uppercase and lowercase converted value, and generates a disjunction $A \mid a$.

109b ⟨*Stack operations* 107c⟩+≡
```
METHOD CharTop(Code: INTEGER);
  VAR
    CharExpr: CharExpression;
    OtherCode: INTEGER;

  ⟨Between local method 110a⟩

  BEGIN
  CharExpr.CREATE (Code);
  Eval.Push (CharExpr);
  IF YLexCfg.IgnoreCase THEN
    ⟨Compute the alternate code OtherCode 110b⟩
    IF OtherCode ≠ 0 THEN
      CharExpr.CREATE (OtherCode);
      Eval.Push (CharExpr);
      OrTop;
      END;
    END;
  END CharTop;
```

110a ⟨Between *local method* 110a⟩≡

```
METHOD Between (Code: INTEGER;
                Ch1, Ch2: CHAR): BOOLEAN;
  BEGIN
  RESULT ← (Code ≥  SYSTEM.ORD(Ch1)) ∧
      (Code ≤  SYSTEM.ORD(Ch2));
  END Between;
```

If the character is an uppercase or lowercase letter, `OtherCode` receives the corresponding character in the other case.

110b ⟨*Compute the alternate code* OtherCode 110b⟩≡

```
IF Between (Code, ‘‘A”, ‘‘Z”) THEN
  OtherCode ← SYSTEM.ORD(SYSTEM.LCASE(SYSTEM.CHR(Code)));
 ELSIF Between (Code, ‘‘a”, ‘‘z”) THEN
  OtherCode ← SYSTEM.ORD(SYSTEM.UCASE(SYSTEM.CHR(Code)));
  END;
```

The `VoidTop` method pushes a void expression onto the stack (See class *Expression. VoidExpression* on page 95). A void expression matches an empty string.

110c ⟨*Stack operations* 107c⟩+≡

```
METHOD VoidTop;
  VAR
    VoidExpr: VoidExpression;
  BEGIN
  VoidExpr.CREATE;
  Eval.Push (VoidExpr);
  END VoidTop;
```

`AnyTop` handles a conveniency shortcut originally provided by Lex. A dot "." in a regular expression matches any character except a newline.

110d ⟨*Stack operations* 107c⟩+≡

```
METHOD AnyTop;
  VAR
    ChSetExpr: CharSetExpression;
    ChSet: CharSet;
  BEGIN
```

`AnyTop` creates a `CharSet` that includes all characters except the newline character.

110e ⟨*Stack operations* 107c⟩+≡

```
ChSet.CREATE;
FOR i ∈ 1 .. YLexCfg.MaxTokenCode-1 | i ≠ NewLine DO
  ChSet.Include (i);
  END;
```

Then, a `CharSetExpression` instance is created with this `CharSet` and pushed onto the evaluation stack.

111a ⟨*Stack operations* 107c⟩+≡
```
ChSetExpr.CREATE (ChSet);
Eval.Push (ChSetExpr);
END AnyTop;
```

When an entire regular expression has been parsed, the stack content must be concatenated in order to produce a single expression. This is achieved by the following code fragment, that calls `ConcatTop` (See page 109) repeatedly, and leaves the resulting expression on the stack.

111b ⟨*Implicit concatenation of the evaluation stack* 111b⟩≡
```
WHILE Eval.Depth > 1 DO
  ConcatTop;
  END;
ASSERT Eval.Depth = 1;
```

The following code fragments summarize the action that is taken when a given token is found in the input. Except for the case of an `Or` operator, the mapping between the operators and the methods that work on the implicit stack is trivial.

The `Or` operator (|) is slightly different, since its syntax is infix instead of being postfix as all other operators listed here.

111c ⟨*Simple operators* 111c⟩≡
```
Star:
  StarTop;
  END;
Or:
  ⟨Handling the Or operator 112a⟩
  END;
Plus:
  PlusTop;
  END;
Void:
  VoidTop;
  END;
Any:
  AnyTop;
  END;
```

The `Or` operator is handled in a rather primitive way: The rest of the line (its right part) must be a valid expression, recognizable using a recursive call to `ParseCode` (See page 105). If it is the case, this expression is pushed onto the stack, and the `OrTop` (See page 108) method will leave the disjunction of this right part with the left part that had been computed before on the stack.

This mechanism is required for this operator only, since it is the only infix operator except for implicit concatenation which is taken care of in an even more brute force way.

112a ⟨*Handling the* `Or` *operator* 112a⟩≡

```
Expr ← ParseCode (Code.SLICE (FromPos+1,
                              Code.SIZE -FromPos-1));
IF Expr ≠ Λ THEN
  Eval.Push (Expr);
  OrTop;
  FromPos ← Code.SIZE;
 ELSE
  ExprError ← TRUE;
  END;
```

The code that takes care of parentheses in a regular expression is amazingly simple: it simply finds the matching closing parentheses, extracts the corresponding slice *without* the parentheses, and performs a recursive call to `ParseCode` (See page 105). It then leaves the resulting expression on its own evaluation stack, and synchronizes the input pointer in such a way that the processing continues after the closing parenthese.

112b ⟨*Handling opening parentheses* 112b⟩≡

```
ToPos ← SearchRParen (Code, FromPos);
IF ToPos < Code.SIZE THEN
  Expr ← ParseCode(Code.SLICE (FromPos+1, ToPos-FromPos-1));
  IF Expr ≠ Λ THEN
    Eval.Push (Expr);
    FromPos ← ToPos;
   ELSE
    ExprError ← TRUE;
    END;
 ELSE
  StdErr.WriteString (‘‘Missing closing parenthesis in expression: ”);
  ExprError ← TRUE;
  END;
```

The code to handle brackets is quite similar: it extracts a slice using the `SearchRBracket` method (See page 107), and transmits the slice to the `HandleBracket` method (See the current page). The difference between plain brackets and escaped brackets has been taken care of before, by the `Codify` method (See page 100).

113a ⟨*Handling opening bracket* 113a⟩≡

```
ToPos ← SearchRBracket (Code, FromPos);
IF ¬ ExprError THEN
  IF ToPos < Code.SIZE THEN
    HandleBracket (Code.SLICE (FromPos+1, ToPos-FromPos-1));
    FromPos ← ToPos;   -- Synchronize on the closing bracket
   ELSE
    StdErr.WriteString (“Missing closing bracket ” +
                        “in expression: ”);
    ExprError ← TRUE;
    END;
  END;
```

`HandleBracket` handles the content of brackets that represent a set of characters. It considers the leading “ˆ” character to negate the set; and the range operator “–” to define character intervals. It creates a `CharSetExpression` (See page 89) and leaves it on the evaluation stack.

113b ⟨`HandleBracket` *local method* 113b⟩≡

```
METHOD HandleBracket(Code: ARRAY OF INTEGER);
  VAR
    CurrentCode, Pos: INTEGER;
    Negative: BOOLEAN;
    ChSet: CharSet;
    ChExpr: CharSetExpression;
  BEGIN
  ChSet.CREATE;
  ⟨Check for negative range 115a⟩
  CurrentCode ← Code_Pos;
  ⟨Loop on the content of Code 114a⟩
  IF ExprError ∨ (Pos < Code.SIZE) THEN
    StdErr.WriteString (“Illegal bracket syntax ” +
                        “in expression [1]”);
    ExprError ← TRUE;
    END;
  IF ¬ ExprError THEN
    ⟨Negate if required 115b⟩
    ⟨Create and leave the CharSetExpression on the evaluation stack 115c⟩
    END;
  END HandleBracket;
```

114a ⟨*Loop on the content of* `Code` 114a⟩≡

```
WHILE ¬ ExprError ∧ (Pos < Code.SIZE)
          ∧ (⟨CurrentCode is valid 114b⟩) DO
  ⟨Handle the current code within the range expression 114c⟩
  Pos ← Pos + 1;
  IF Pos < Code.SIZE THEN
    CurrentCode ← Code_{Pos};
   ELSE
    CurrentCode ← 0;
    END;
  END;
```

114b ⟨`CurrentCode` *is valid* 114b⟩≡

```
((CurrentCode < YLexCfg.MaxTokenCode) ∨
 (CurrentCode = Range) ∨
 (CurrentCode = LParen) ∨
 (CurrentCode = RParen))
```

114c ⟨*Handle the current code within the range expression* 114c⟩≡

```
IF CurrentCode = Range THEN
  ⟨Handle the case where the current code is Range 114d⟩
 ELSIF CurrentCode < YLexCfg.MaxTokenCode THEN
  ⟨Handle the case where the current code is not Range 114e⟩
  END;
```

Range handling is a rather brute force piece of code. If the current token is the `Range` operator, and if the next and previous tokens are valid simple characters (valid in the sense of ranges within the array `Code`), if these characters are sorted properly, that is, the first one is smaller or equal to the second one, then include all the characters between these two limits.

114d ⟨*Handle the case where the current code is* `Range` 114d⟩≡

```
IF (Pos > 0) ∧ (Pos + 1 < Code.SIZE) ∧
    (Code_{Pos-1} < YLexCfg.MaxTokenCode) ∧
    (Code_{Pos+1} < YLexCfg.MaxTokenCode) ∧
    (Code_{Pos-1} ≤ Code_{Pos+1}) THEN
  FOR i ∈ Code_{Pos-1} .. Code_{Pos+1} DO
    ChSet.Include (i);
    END;
 ELSE
  ExprError ← TRUE;
  END;
Pos ← Pos + 1;
```

If the current code is not `Range`, simply add the code to the character set. Of course, this is redundant if the current code is followed by `Range`, but who cares, really...

114e ⟨*Handle the case where the current code is not* `Range` 114e⟩≡

```
ChSet.Include (Code_{Pos});
```

If the bracket expression starts with the `Not` code, indicating that the set must be negated, the `Negative` local variable is set to true, and the input pointer is advanced.

115a ⟨*Check for negative range* 115a⟩≡

```
IF Code0 = Not THEN
  Negative ← TRUE;
  Pos ← 1;
  END;
```

Before exiting the method, if the bracket expression started with a `Not` code, the resulting character set is negated, achieving the purpose of the negation operator.

115b ⟨*Negate if required* 115b⟩≡

```
IF Negative THEN
  ChSet.Negate;
  END;
```

Now that we know that no error has been detected, we can safely create our `CharSetExpression` `ChExpr` using `ChSet` (which basically is a bitset), and leave it on the evaluation stack.

115c ⟨*Create and leave the* `CharSetExpression` *on the evaluation stack* 115c⟩≡

```
ChExpr.CREATE (ChSet);
Eval.Push (ChExpr);
```

The following code summarizes the action that is taken if some token are found in the input. In a well-formed regular expression, such tokens should never occur at this level. For instance, the right parenthese (`RParen`) should be recognized and parsed when analyzing the left parenthese. Similarly, the range operator "-" should never occur outside a range expression which is handled when encountering a opening bracket.

116a ⟨*Handling error tokens* 116a⟩≡
```
RParen:
  StdErr.WriteString (‘‘Missing opening parenthesis in ” +
                      ‘‘expression: ”);
  ExprError ← TRUE;
  END;
RBracket:
  StdErr.WriteString (‘‘Missing opening bracket in ” +
                      ‘‘expression: ”);
  ExprError ← TRUE;
  END;
Not:
  StdErr.WriteString (‘‘Illegal Not operator in ” +
                      ‘‘expression: ”);
  ExprError ← TRUE;
  END;
Range:
  StdErr.WriteString (‘‘Illegal Range operator in ”+
                      ‘‘expression: ”);
  ExprError ← TRUE;
  END;
```

116b ⟨`ExpressionParser` *class implementation* 99b⟩+≡
```
⟨Parse method implementation 117a⟩
```

116c ⟨`Parse` *method definition for* `ExpressionParser` 116c⟩≡
```
METHOD Parse(Source: ARRAY OF CHAR;
       Spec: LexSpecification): Expression;
```

The `Parse` method is a rather simple combination of `Codify`, that produces a tokenized version of the character string `Source`, and `ParseCode` that takes that tokenized version and produces an equivalent expression tree.

117a ⟨`Parse` *method implementation* 117a⟩≡

```
METHOD Parse(Source: ARRAY OF CHAR;
                Spec: LexSpecification): Expression;
  VAR
    Code: ARRAY OF INTEGER;
  BEGIN
  ExprError ← FALSE;
  ⟨Tokenize the regular expression in source form 117c⟩
  RESULT ← ParseCode (Code);
  IF ExprError THEN
    StdErr.WriteLine (Source);
    YLexCfg.SetError (TRUE);
    END;
  END Parse;
```

117b ⟨`ExpressionParser` *class implementation* 99b⟩+≡

```
⟨MakeTokenClass method implementation 118⟩

END ExpressionParser;
```

The `Codify` method (See page 100) is called to build a tokenized version of the regular expression. The specification `Spec` is also given as parameter so that variable references can be resolved. For instance, when considering a token definition of the form `Number   {DIGIT}+`, `Spec` will be used to find the definition of `DIGIT` using the `FindVariable` method (See page 63).

117c ⟨*Tokenize the regular expression in source form* 117c⟩≡

```
Code ← Codify (Source, Spec);
```

`MakeTokenClass` builds a (potentially huge) disjunction of `Expressions` it receives as parameters, and returns this disjunction. Its name comes from its most common usage: this method is used when defining a token class as being a set of tokens. Its attached expression is the disjunction of the expressions attached to these tokens.

117d ⟨`MakeTokenClass` *method definition* 117d⟩≡

```
METHOD MakeTokenClass(Arr: ARRAY OF Expression): Expression;
```

118 ⟨MakeTokenClass *method implementation* 118⟩≡

```
METHOD MakeTokenClass(Arr: ARRAY OF Expression): Expression;
  VAR
    OrExpr: OrExpression;
  BEGIN
  FOR Expr ∈ Arr DO
    IF RESULT = Λ THEN
      RESULT ← Expr;
     ELSE
      OrExpr.CREATE (RESULT, Expr);
      RESULT ← OrExpr;
      END;
    END;
  END MakeTokenClass;
```

4.6 Automaton

This module is just a wrapper around a NFA (Non-deterministic finite automaton). It is made of a couple of `Node` (See page 137), identifying the entry point and the exit point of the NFA.

⟨*Definition* ⟩≡
```
DEFINITION MODULE Automaton;

IMPORT Couple;
FROM NodeSet IMPORT NodeSet;
FROM Node IMPORT Node;

  CLASS Automaton;
    INHERITS Couple(Node);

    METHOD First: Node;
    METHOD Last: Node;
  END Automaton;

END Automaton;
```

This class inherits from one the simplest possible parameterized classes provided by the YAFL standard library, named `Couple`, that implements couples (amazing, ain't it ?). `Couple` provides two methods: `One` that returns the first element of the couple, `Two` returns the other element. YAFL

⟨*Implementation* ⟩≡
```
IMPLEMENTATION MODULE Automaton;

  CLASS Automaton;
    INHERITS Couple(Node);

    METHOD First: Node;
      BEGIN
      RESULT ← One;
      END First;

    METHOD Last: Node;
      BEGIN
      RESULT ← Two;
      END Last;

  END Automaton;

END Automaton;
```

4.7 CharSet

YAFL

The `CharSet` class implements sets of character. It is implemented as a simple wrapper around bitmaps (as implemented by the `BitMap` class as provided by the YAFL standard library), and all its operations are translated trivially into bitmaps primitives.

⟨*Definition* ⟩≡

```
DEFINITION MODULE CharSet;

  CLASS CharSet;

    REDEFINE METHOD CREATE;

    ⟨Include method definition 121b⟩
    ⟨Includes method definition 121d⟩
    ⟨Or method definition 121f⟩
    ⟨Negate method definition 121h⟩

  END CharSet;

END CharSet;
```

⟨*Implementation* ⟩≡

```
IMPLEMENTATION MODULE CharSet;

IMPORT BitMap;
IMPORT SYSTEM;
IMPORT YLexCfg;

  CLASS CharSet;

    VAR
      Data: BitMap;

    ⟨Creating a CharSet 121a⟩
    ⟨Include method implementation 121c⟩
    ⟨Includes method implementation 121e⟩
    ⟨Or method implementation 121g⟩
    ⟨Negate method implementation 122⟩
  END CharSet;

END CharSet;
```

When creating the `CharSet`, one must create its embedded `BitMap` large enough to include all characters (See method *YLexCfg.YLexCfg.MaxTokenCode* on page 178).

121a ⟨*Creating a* `CharSet` 121a⟩≡
```
REDEFINE METHOD CREATE;
  BEGIN
  Data.CREATE (YLexCfg.MaxTokenCode);
  END CREATE;
```

The `Include` method adds a given character in the set. The post-condition described here indicates that if a `CharSet` includes `Code` (By means of the `Includes` method (See this page)), including it again will not alter the object's internal structure in any way. Besides, in any case, after adding `Code` to a `CharSet`, it will include `Code`.

121b ⟨`Include` *method definition* 121b⟩≡
```
METHOD Include(Code: INTEGER);
  POST
    OLD Includes(Code) ⇒ NO_CHANGE,
    Includes(Code);
```

121c ⟨`Include` *method implementation* 121c⟩≡
```
METHOD Include(Code: INTEGER);
  BEGIN
  Data.Set(Code);
  END Include;
```

121d ⟨`Includes` *method definition* 121d⟩≡
```
METHOD Includes(Code: INTEGER): BOOLEAN;
```

121e ⟨`Includes` *method implementation* 121e⟩≡
```
⟨Includes method definition 121d⟩
BEGIN
ASSERT Code ≥ 0;
RESULT ← Data_Code;
END Includes;
```

121f ⟨`Or` *method definition* 121f⟩≡
```
METHOD Or(Other: CharSet);
```

121g ⟨`Or` *method implementation* 121g⟩≡
```
⟨Or method definition 121f⟩
  BEGIN
  Data.Or (Other.Data);
  END Or;
```

121h ⟨`Negate` *method definition* 121h⟩≡
```
METHOD Negate;
```

122 ⟨`Negate` *method implementation* 122⟩≡
  ⟨`Negate` *method definition* 121h⟩

```
BEGIN
Data.Negate;
END Negate;
```

4.8 DfaAutomaton

`DfaAutomaton` is a wrapper class, that takes a `LexSpecification` (See page 63) and represents the equivalent deterministic finite state automaton (DFA). To do so, it uses the non-deterministic finite automaton (NFA) which is available from the `LexSpecification` and uses very common (and rather inefficient) algorithms to deduce the DFA.

⟨*Definition* ⟩≡
```
DEFINITION MODULE DfaAutomaton;

FROM Node IMPORT Node;
FROM Token IMPORT LexSpecification;

  CLASS DfaAutomaton;
    REDEFINE METHOD CREATE (StartNode: Node;
                            Spec: LexSpecification);
    ⟨GenerateCode method definition 124b⟩
    ⟨DumpStatistics method definition 132b⟩
  END DfaAutomaton;

END DfaAutomaton;
```

⟨*Implementation* ⟩≡
```
IMPLEMENTATION MODULE DfaAutomaton;

⟨Import list 133c⟩

  CLASS DfaAutomaton;

    ⟨DfaAutomaton attributes 124a⟩

    ⟨CREATE method implementation 129a⟩
    ⟨GenerateAutomaton method 131c⟩
    ⟨GenerateAncillaryMethods method 132a⟩
    ⟨GenerateKeywords method implementation 128b⟩
    ⟨GenerateTokenImage method implementation 128d⟩
    ⟨GenerateCode method implementation 124c⟩

    ⟨DumpStatistics method implementation 132c⟩

  END DfaAutomaton;

END DfaAutomaton;
```

TheNodeSets is a list of NodeSets (See page 151). Each of these NodeSet is a set of states in the NFA, and correspond to a valid DFA state. In other words, TheNodeSets is the list of valid states in the DFA.

TheSpec is a reference to the LexSpecification (See page 63) that was given at creation time.

124a ⟨DfaAutomaton *attributes* 124a⟩≡

```
VAR
  TheNodeSets: NodeSetList;
  TheSpec: LexSpecification;
```

The code generation method for a DFA is the last phase depicted in figure 4.2 on page 44. It does not include much intelligence of its own. It merely reads *skeleton files* (See page 212) which contain the common parts among all lexical analyzer to generate, and produces output files made of these skeletons where *patterns* are replaced by ad hoc subparts, depending on the lexical analyzer one generates.

124b ⟨GenerateCode *method definition* 124b⟩≡

```
METHOD GenerateCode (ModuleName: ARRAY OF CHAR): BOOLEAN;
```

124c ⟨GenerateCode *method implementation* 124c⟩≡

```
METHOD GenerateCode (ModuleName: ARRAY OF CHAR): BOOLEAN;
  ⟨Filename normalization method 124e⟩
  ⟨ExpandSkeleton method implementation 125a⟩

  BEGIN
  ⟨Renumber TheNodeSets 124d⟩
  RESULT ←
    ExpandSkeleton(‘‘iskel.lex”,NormFName(ModuleName)+‘‘.yy”) ∧
    ExpandSkeleton(‘‘dskel.lex”,NormFName(ModuleName)+‘‘.dy”);
  END GenerateCode;
```

Since the generated lexer only deals with numbers rather than with references, the states of the automaton must be numbered, by means of the Renumber method (See page 171).

124d ⟨*Renumber* TheNodeSets 124d⟩≡

```
TheNodeSets.Renumber;
```

NormFName takes a module name and returns the proper radix for corresponding filenames according to the YAFL convention: truncate to the 8 first characters, and convert to lowercase characters only.

124e ⟨*Filename normalization method* 124e⟩≡

```
METHOD NormFName (a: ARRAY OF CHAR): ARRAY OF CHAR;
  BEGIN
  RESULT ← String.LowerCase(a.CLONE);
  IF RESULT.SIZE > 8 THEN
    RESULT ← RESULT.SLICE(0, 8);
    END;
  END NormFName;
```

`ExpandSkeleton` reads a skeleton file, copies it onto a target file, except for codes that are made of the '$' character followed by a letter that are replaced by elements related to the DFA. By defining ad hoc skeleton files, one can adapt `lexyt` to a specific case without having to go through large amounts of obfuscated sources.

125a ⟨`ExpandSkeleton` *method implementation* 125a⟩≡

```
METHOD ExpandSkeleton(SourceFName,
                      TargetFName: ARRAY OF CHAR): BOOLEAN;
  VAR
    Output: VersionedOutputStream;
    Input: InputStream;
    Ch: CHAR;
  BEGIN
  ⟨Open Input file 125b⟩
  IF Input.ErrorCode ≠ Input.NoError THEN
    StdErr.WriteLine (“Cannot open file ” + SourceFName +
                      “ for input”);
   ELSE
    ⟨Input is open, proceed 125c⟩
    END;
  END ExpandSkeleton;
```

When opening the skeleton files, we first try to open it in the current directory, so that one can use a project-specific version of these skeleton files. If no such version can be found, we try to find the skeleton file in the directory referred to by the `YAFL_HOME` environment variable, which is used by all YAFL tools for similar purposes. YAFL

125b ⟨*Open* `Input` *file* 125b⟩≡

```
Input.CREATE;
Input.Open (SourceFName, Input.ReadAccess);
IF Input.ErrorCode ≠ Input.NoError THEN
  Input.Open (SYSTEM.GetEnvironment (“YAFL_HOME”) +
              SYSTEM.DirSeparator + SourceFName,
                Input.ReadAccess);
  END;
```

125c ⟨`Input` *is open, proceed* 125c⟩≡

```
IF YLexCfg.Verbose THEN
  StdOut.WriteLine (“Generating target file ” + TargetFName +
                    “ based on skeleton file ” + SourceFName);
  END;
```

`Output` is a `VersionedOutputStream`. This class is a specialized `OutputStream` that will not replace an existing file unless it introduces a change. This feature proves useful, by avoiding cascading useless compilations within a `makefile` that occur when a file's timestamp is changed without changing its content. YAFL

126a ⟨`Input` *is open, proceed* 125c⟩+≡

```
Output.CREATE;
Output.AttemptCreate (TargetFName, TargetFName + ‘‘$”);
WHILE ¬ Input.Eof DO
  Ch ← Input.ReadChar;
  IF Ch = ‘‘$” THEN
    ASSERT ¬ Input.Eof;
    Ch ← Input.ReadChar;
    ⟨Depending on the code following $, take appropriate action 126b⟩
   ELSE
    Output.WriteChar (Ch);
    END;
  END;
Input.Close;
Output.Close;
RESULT ← TRUE;
```

First, handle codes for simple issues such as date and version number (See page 45).

126b ⟨*Depending on the code following $, take appropriate action* 126b⟩≡

```
CASE Ch OF
  ‘‘M”:
    Output.WriteString (ModuleName);
    END;
  ‘‘D”:
    Output.WriteString (IntConversions.IntToString(Today.Day,0) +
            ‘‘/” + IntConversions.IntToString(Today.Month,0) +
            ‘‘/” + IntConversions.IntToString(Today.Year,0));
    END;
  ‘‘V”:
    Output.WriteString (LexYt.Version);
    END;
```

Then, more useful things: generating the automaton (See page 131) ...

126c ⟨*Depending on the code following $, take appropriate action* 126b⟩+≡

```
‘‘A”:
  GenerateAutomaton (Output);
  END;
```

generating the code to initialize the results array, indexed by DFA state, and which indicates the tokens have been recognized when reaching a DFA state (See page 172).

127a ⟨*Depending on the code following $, take appropriate action* 126b⟩+≡
```
  ''I":
    TheNodeSets.GenerateResultArrays (Output);
    END;
```

generating the body of the `TokenImage` method (See method *DfaAutomaton.DfaAutomaton.GenerateTokenImage* on the next page) . . .

127b ⟨*Depending on the code following $, take appropriate action* 126b⟩+≡
```
  ''T":
    GenerateTokenImage (Output);
    END;
```

generating the symbolic constants (See the following page) . . .

127c ⟨*Depending on the code following $, take appropriate action* 126b⟩+≡
```
  ''K":
    GenerateKeywords (Output);
    END;
```

if required, generating the ancillary methods (See page 132). Otherwise, the `''$S''` pattern in the skeleton file is ignored . . .

127d ⟨*Depending on the code following $, take appropriate action* 126b⟩+≡
```
  ''S":
    IF YLexCfg.UseAncillaryMethods THEN
      GenerateAncillaryMethods (Output);
      END;
    END;
```

$B expands to a reference to `TheBackTrackLevel` variable (See page 218) if the lexer's backtracking level is dynamic, and to the constant `BACKTRACKLEVEL` if this backtracking level is statically set when producing the lexer.

127e ⟨*Depending on the code following $, take appropriate action* 126b⟩+≡
```
  ''B":
    IF YLexCfg.DynamicBackTrackLevel THEN
      Output.WriteString (''TheBackTrackLevel");
     ELSE
      Output.WriteString (''BACKTRACKLEVEL");
      END;
    END;
```

One can also use a double $ sign to paste a single literal $ sign in the target file.

127f ⟨*Depending on the code following $, take appropriate action* 126b⟩+≡
```
  ''$":
    Output.WriteChar(''$");
    END;
```

If the character that follows $ is not recognized, the $ is simply ignored, and the following character is pasted as is in the target file.

128a ⟨*Depending on the code following $, take appropriate action* 126b⟩+≡

```
ELSE
  Output.WriteChar (Ch);
  END;
```

`GenerateKeywords` output is meant to be pasted somewhere in a definition module. It describes some of the generated lexer's properties, such as the backtracking level. It also generates one YAFL constant declarations for each token of the specification.

128b ⟨`GenerateKeywords` *method implementation* 128b⟩≡

```
METHOD GenerateKeywords (Output: OutputStream);
  BEGIN
  Output.WriteString (‘‘  BACKTRACKLEVEL = ”);
  Output.WriteInt (YLexCfg.BackTrackLevel, 0);
  Output.WriteLine (‘‘;”);
  Output.WriteString (‘‘  DYNAMICBACKTRACKLEVEL = ”);
  IF YLexCfg.DynamicBackTrackLevel THEN
    Output.WriteLine (‘‘TRUE;”);
   ELSE
    Output.WriteLine (‘‘FALSE;”);
    END;
  ⟨Generate constant declarations for the tokens 128c⟩

  Output.WriteString (‘‘      LASTTOKEN = FIRSTTOKEN + ” );
  Output.WriteInt (TheSpec.Size, 0);
  Output.WriteLine (‘‘;”);
  END GenerateKeywords;
```

The constants are actually produced by the `GenerateConstDecl` method (See page 57).

128c ⟨*Generate constant declarations for the tokens* 128c⟩≡

```
FOR s ∈ TheSpec DO
  s.GenerateConstDecl (Output);
  END;
```

Similarly, the responsibility of the generation of the body of the `TokenImage` method is deferred to the tokens themselves (See method *Token.Token.GenerateTokenImage* on page 57).

128d ⟨`GenerateTokenImage` *method implementation* 128d⟩≡

```
METHOD GenerateTokenImage (Output: OutputStream);
  BEGIN
  FOR s ∈ TheSpec DO
    s.GenerateTokenImage (Output);
    END;
  END GenerateTokenImage;
```

129a ⟨CREATE *method implementation* 129a⟩≡
```
REDEFINE METHOD CREATE (StartNode: Node;
                        Spec: LexSpecification);
  VAR
  ⟨CREATE locals 129b⟩
```

129b ⟨CREATE *locals* 129b⟩≡
```
Exist, StartSet, New, Current: NodeSet;
ValidTran: CharSet;
```

129c ⟨CREATE *method implementation* 129a⟩+≡
```
BEGIN
IF YLexCfg.Verbose THEN
  StdOut.WriteLine ("Creating Dfa Automaton");
  StdOut.WriteLn;
  END;
TheSpec ← Spec;
TheNodeSets.CREATE;
WorkStack.CREATE;
```

As common with algorithms that compute closures or related data structures, we have the choice between two implementation strategies: recursion or keeping track of the items to visit. We choose the latter; and keep track of the items to visit in `WorkStack`.

129d ⟨CREATE *locals* 129b⟩+≡
```
WorkStack: Stack(NodeSet);
```

First, `StartSet` is initialized as the ϵ-closure of the node `StartNode` (See method *Node.Node.GetEpsClosure* on page 139). It is the set of states of the NFA that can be reached without consuming a token. Hence, `StartSet` represents the first state of our DFA, and is appended to `NodeSets`.

It is pushed onto `WorkStack` so that the loop can extract it and visit all DFA states that are reachable starting at `StartSet`.

129e ⟨CREATE *method implementation* 129a⟩+≡
```
StartSet ← StartNode.GetEpsClosure;
TheNodeSets.Append (StartSet);
WorkStack.Push (StartSet);
```

Now that the start state has been pushed onto `WorkStack`, we loop as long as we have something to analyze. Of course, analyzing a `NodeSet` might require that other `NodeSet`s are analyzed as well, in which case they are pushed onto `WorkStack`.

130a ⟨`CREATE` *method implementation* 129a⟩+≡
```
  WHILE WorkStack.Depth > 0 DO
    ⟨Dump status information if required 130b⟩
    Current ← WorkStack.Pop;
    ⟨Make sure that Current looks kosher 130c⟩
    ValidTran ← Current.ValidTransitionsChars;
    FOR Char ∈ 0 .. YLexCfg.MaxTokenCode DO
      ⟨Check transition for Char 130e⟩
      END;
    END;
  END CREATE;
```

130b ⟨*Dump status information if required* 130b⟩≡
```
  IF YLexCfg.Verbose THEN
    StdOut.WriteInt (WorkStack.Depth, 5);
    StdOut.WriteString (" waiting's ");
    StdOut.WriteInt (TheNodeSets.Size, 5);
    StdOut.WriteString (" dfa's ");
    StdOut.WriteInt (SYSTEM.Allocated, 8);
    StdOut.WriteLine (" mem");
    END;
```

`Current` cannot be void, of course . . .

130c ⟨*Make sure that* `Current` *looks kosher* 130c⟩≡
```
  ASSERT Current ≠ Λ;
```

Besides, it must referenced in `TheNodeSet` as one of the already recognized states of our automaton (See method *NodeSet.NodeSetList.Includes* on page 172).

130d ⟨*Make sure that* `Current` *looks kosher* 130c⟩+≡
```
  ASSERT TheNodeSets.Includes(Current) = Current;
```

Finally, it must be closed, that is, it must be equal to its own ϵ-closure (See method *NodeSet.NodeSet.IsClosed* on page 157). ASSERT Current.IsClosed;

130e ⟨*Check transition for* `Char` 130e⟩≡
```
  IF ValidTran.Includes (Char) THEN
    New ← Current.OnCharClosure (Char);
    Exist ← TheNodeSets.Includes(New);
    IF Exist ≠ Λ THEN
      ⟨No new DFA state must be created 131a⟩
     ELSE
      ⟨A new DFA state must be created 131b⟩
      END;
    END;
```

If `Exist` is not `VOID`, no new DFA state must be created, but `Current` must be connected to `Exist` on transition `Char`. In other words, when in state `Current` and receiving the code `Char`, the automaton must move to state `Exist` instead of `New`.

131a ⟨*No new DFA state must be created* 131a⟩≡

```
Current.Connect(Char, Exist);
```

If there is no DFA state equivalent to `New` in `NodeSets`, then a new DFA state must be created, and attached to `Current` on transition `CHR(Char)`. Hence, it is appended to `NodeSets`, and pushed onto `WorkStack` so that further states that are reachable from this new state can be analyzed.

131b ⟨*A new DFA state must be created* 131b⟩≡

```
Current.Connect(Char, New);
WorkStack.Push (New);
TheNodeSets.Append (New);
ASSERT New.IsClosed;
```

Depending on a configuration flag (See method *YLexCfg.YLexCfg.UseAncillaryMethods* on page 178), `lexyt` generates the automaton inline (that is, with the ad hoc logic pasted in each state) or using external methods, in which case the action to be taken in each state is to call the corresponding method.

131c ⟨`GenerateAutomaton` *method* 131c⟩≡

```
METHOD GenerateAutomaton(Output: OutputStream);
  BEGIN
  FOR i ∈ 0 .. TheNodeSets.Size - 1 DO
    Output.WriteInt (i, 0);
    Output.WriteLine (":");
    IF YLexCfg.UseAncillaryMethods THEN
      Output.WriteLine ("METH" +
                        IntConversions.IntToString (i, 0) +
                        " (Ch);");
     ELSE
      TheNodeSets_i.GenerateCode (Output);
      END;
    Output.WriteLine ("END;");
    END;
  END GenerateAutomaton;
```

GenerateAncillaryMethods generates the methods which include the logic to be followed for every state in the automaton.

132a ⟨GenerateAncillaryMethods *method* 132a⟩≡

```
METHOD GenerateAncillaryMethods(Output: OutputStream);
  BEGIN
  FOR i ∈ 0 .. TheNodeSets.Size − 1 DO
    Output.WriteLine (“METHOD METH” + IntConversions.IntToString (i, 0) +
                      “ (Ch: INTEGER);”);
    Output.WriteLine (“BEGIN”);
    TheNodeSets_i.GenerateCode (Output);
    Output.WriteLine (“END METH” + IntConversions.IntToString (i, 0) + “;”)
    Output.WriteLn;
    END;
  END GenerateAncillaryMethods;
```

132b ⟨DumpStatistics *method definition* 132b⟩≡

```
METHOD DumpStatistics;
```

132c ⟨DumpStatistics *method implementation* 132c⟩≡

```
METHOD DumpStatistics;
  VAR
    x, Avg: REAL;
    Tot, Count, j, Min, Max, Sum, Sum2: INTEGER;
  BEGIN
  Min ← 9999999;
  Max ← -9999999;
  Count ← TheNodeSets.Size;
  FOR Node ∈ TheNodeSets DO
    j ← Node.UsedBits;
    ⟨Accumulate the number of bits j 133a⟩
    END;
  StdOut.WriteString (“Min: ”);
  StdOut.WriteInt (Min, 0);
  StdOut.WriteString (“ Max: ”);
  StdOut.WriteInt (Max, 0);
  StdOut.WriteString (“ Avg: ”);
  Avg ← SYSTEM.FLOAT(Sum)/SYSTEM.FLOAT(Count);
  StdOut.WriteReal (Avg, 8, 2);
  StdOut.WriteString (“ StdDev: ”);
  x ← SYSTEM.FLOAT(Sum2)/SYSTEM.FLOAT(Count) − (Avg * Avg);
  ASSERT x ≥ 0.0;
  StdOut.WriteReal (Math.Sqrt(x), 8, 2);
  StdOut.WriteLn;
  ⟨Display bitmap paging stats 133b⟩
  END DumpStatistics;
```

To produce the various parameters, we need the sum of the number of bits in `Sum`, and the sum of the squares of the number of bits in `Sum2`.

133a ⟨*Accumulate the number of bits* `j` 133a⟩≡

```
IF j < Min THEN
  Min ← j;
  END;
IF j > Max THEN
  Max ← j;
  END;
Sum ← Sum + j;
Sum2 ← Sum2 + (j*j);
```

133b ⟨*Display bitmap paging stats* 133b⟩≡

```
StdOut.WriteString (“Total referred pages:”);
FOR Node ∈ TheNodeSets DO
  Tot ← Tot + Node.UsedPages;
  END;
StdOut.WriteInt (Tot, 0);
StdOut.WriteLn;
StdOut.WriteString (“Dictionary pages:”);
StdOut.WriteInt (PageDictionary.Stored, 0);
StdOut.WriteLn;
```

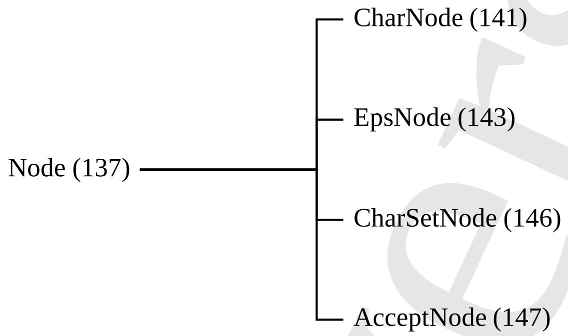
133c ⟨*Import list* 133c⟩≡

```
IMPORT CharSet;
IMPORT YLexCfg;
FROM YLex IMPORT LexYt;
FROM Conversions IMPORT IntConversions;
FROM DateTime IMPORT Today;
IMPORT Math;
FROM Node IMPORT NodeDictionary;
FROM NodeSet IMPORT NodeSetList, NodeSet;
IMPORT Stack;
FROM Stats IMPORT StatCollector;
FROM Streams IMPORT StdOut, OutputStream, VersionedOutputStream,
                    InputStream, StdErr;
IMPORT String;
IMPORT SYSTEM;
FROM DuraBitMap IMPORT PageDictionary;
```

4.9 Node

The module `Node` describes the classes that are required to build the non-deterministic finite-state automaton (NFA) that will be used as basis to build its equivalent deterministic counterpart.

Figure 4.11 Inheritance tree for `Node`



⟨*Definition* ⟩≡
```
DEFINITION MODULE Node;

IMPORT CharSet;
FROM NodeSet IMPORT NodeSet;
FROM Token IMPORT Token;

   ⟨NodeDictionary class definition 135a⟩
   ⟨Node base class definition 138⟩
   ⟨Node-derived classes definition 141⟩

END Node;
```

⟨*Implementation* ⟩≡
```
IMPLEMENTATION MODULE Node;

FROM Streams IMPORT StdOut;

   ⟨NodeDictionary class 135b⟩
   ⟨Node base class 139a⟩
   ⟨Node-derived classes 142⟩

END Node;
```

135a ⟨`NodeDictionary` *class definition* 135a⟩≡

```
ONCE CLASS NodeDictionary;
  REDEFINE METHOD CREATE;
  METHOD Get(Nr: INTEGER): Node;
  METHOD High: INTEGER;
  METHOD Count: INTEGER;
END NodeDictionary;
```

135b ⟨`NodeDictionary` *class* 135b⟩≡

```
ONCE CLASS NodeDictionary;

  ⟨NodeDictionary local declarations 135c⟩
  ⟨NodeDictionary CREATE 136a⟩

  ⟨Get method 137⟩
  ⟨Enter method 136c⟩

  METHOD High: INTEGER;
    BEGIN
    RESULT ← NextAvailableIndex;
    END High;

  METHOD Count: INTEGER;
    BEGIN
    FOR n ∈ NodeArray | n ≠ Λ DO
      RESULT ← RESULT + 1;
      END;
    END Count;

END NodeDictionary;
```

`InitialSize` indicates the original size of the `NodeDictionary`, but its actual size is dynamic. It will grow by chunks of `InitialSize` elements.

135c ⟨*NodeDictionary local declarations* 135c⟩≡

```
CONST
  InitialSize = 2048;
```

`NodeArray` denotes the array where the nodes will be stored at a position that matches they number. `NextAvailableIndex` denotes the highest index where elements are stored in `NodeArray`.

136a ⟨*NodeDictionary CREATE* 136a⟩≡
```
  REDEFINE METHOD CREATE;
    BEGIN
    BASE;    -- Dummy, since this class has no base class,
             -- just in case an inheritance relationship
             -- is added afterwards to be sure we don't
               -- forget the call to the base class's
               -- CREATE method.
    NodeArray.CREATE (InitialSize);
    END CREATE;
```

136b ⟨*NodeDictionary local declarations* 135c⟩+≡
```
  VAR
    NodeArray: ARRAY OF Node;
    NextAvailableIndex: INTEGER;
```

The `Enter` method enters a `Node` into the `NodeDictionary`, at index `Nr`. `NodeArray` is resized if required.

136c ⟨`Enter` *method* 136c⟩≡
```
  METHOD Enter(Nr: INTEGER;
               TheNode: Node);
    VAR
      ToAdd: ARRAY OF Node;
    BEGIN
    ⟨Consider resizing 136d⟩
    NodeArray_Nr ← TheNode;
    ⟨Update NextAvailableIndex if required 136e⟩
    END Enter;
```

136d ⟨*Consider resizing* 136d⟩≡
```
  WHILE Nr ≥ NodeArray.SIZE DO
    IF ToAdd = Λ THEN
      ToAdd.CREATE (InitialSize);
      END;
    NodeArray ← NodeArray + ToAdd;
    END;
```

136e ⟨*Update* `NextAvailableIndex` *if required* 136e⟩≡
```
  IF Nr ≥ NextAvailableIndex THEN
    NextAvailableIndex ← Nr + 1;
    END;
```

137 ⟨`Get` *method* 137⟩≡

```
METHOD Get(Nr: INTEGER): Node;
  BEGIN
  ASSERT Nr ≥ 0;
  IF Nr < NodeArray.SIZE THEN
    RESULT ← NodeArray_Nr;
    END;
  END Get;
```

`Node` is the base class for all nodes of the NFA. It centralizes the common or default behaviour of all these nodes. Derived classes will then be able to redefine one or two methods, inheriting from the otherwise acceptable default behaviour.

Node

All $Node$s can be connected to another $Node$, implemented as a reference (ρ) (See page 979) in the $TheConnected$ attribute. This reference can be a null reference (Λ). The $\epsilon - Node$ class (See page 143) has an additionnal outgoing connection to another $Node$.
By using the $\downarrow$ operator, $TheConnected$ is specified as referring to a $Node$, *or to one of its derived classes*.

$$TheConnected : \rho \downarrow Node$$

$OnCharClosure$ returns the set of $Node$s that can be reached by consuming an element of $Alphabet$ (See page 13). This set of $Node$s is implemented by the $NodeSet$ class (See page 150).

$OnCharClosure : Alphabet \mapsto NodeSet$ (See page 150)

$$\forall a \in Alphabet \bullet OnCharClosure(a).IsClosed = \text{true}$$

$\epsilon - closure$ returns the set of $Nodes$ that can be reached starting at $self$ without consuming a character at all. This set is implemented by the $NodeSet$ class. Quite logically, $\epsilon - closure$ includes $self$ (which is the ObjectZ equivalent of YAFL's THIS.

$\epsilon - closure : NodeSet$ (See page 150)

$$self \in \epsilon - closure.Data$$

$ValidChars$ returns the set of symbols for which there is a valid outgoing transition from the $Node$ denoted by $self$. These symbols are defined by the $Alphabet$ type (See page 13).

$$ValidChars : \mathbb{P}\, Alphabet$$

$$ValidChars = \{x \in Alphabet \mid OnCharClosure(x) \neq \phi\}$$

138 ⟨Node *base class definition* 138⟩≡

```
CLASS Node;
   REDEFINE METHOD CREATE;
   METHOD Nr: INTEGER;

   ⟨Node connection methods 140a⟩

   METHOD AccumulateOnCharClosure (Code: INTEGER;
                                   TheSet: NodeSet);
```

```
    METHOD EpsClosure (TheSet: NodeSet);
    METHOD AccumulateValidChars (TheCharSet: CharSet);

    METHOD GetEpsClosure: NodeSet;
  END Node;
```

139a ⟨Node *base class* 139a⟩≡

```
  CLASS Node;

    VAR
      TheNr: INTEGER;

    REDEFINE METHOD CREATE;
      VAR
        Counter: ONCE INTEGER;
      BEGIN
      TheNr ← Counter;
      NodeDictionary.Enter (TheNr, THIS);
      Counter ← Counter + 1;
      END CREATE;

    METHOD Nr: INTEGER;
      BEGIN
      RESULT ← TheNr;
      END Nr;

    ⟨GetEpsClosure methods implementation 139b⟩
    ⟨Node connection management 140b⟩
    ⟨Methods to be redefined by Node-derived classes 140c⟩

  END Node;
```

This method is a simple wrapper around `EpsClosure` (See the facing page) and uses an assertion to ensure that the so computed closure is indeed a closure.

139b ⟨GetEpsClosure *methods implementation* 139b⟩≡

```
  METHOD GetEpsClosure: NodeSet;
    BEGIN
    RESULT.CREATE;
    EpsClosure(RESULT);
    ASSERT RESULT.EpsClosure.IsEqual (RESULT);
    END GetEpsClosure;
```

Every Node can be connected to another Node instance, by using the Connect method. This connection can then be read using the Connected method. Besides, the EpsNode class (See page 143) allows an additional outgoing non-deterministic connection.

140a ⟨*Node connection methods* 140a⟩≡
```
METHOD Connect (Other: Node);
METHOD Connected: Node;
  POST
    NO_CHANGE;
```

140b ⟨*Node connection management* 140b⟩≡
```
VAR
  TheConnected: Node;

METHOD Connect (Other: Node);
  BEGIN
  TheConnected ← Other;
  END Connect;

METHOD Connected: Node;
  BEGIN
  RESULT ← TheConnected;
  END Connected;
```

The following implementations are minimal, these methods must be redefined by Node-derived classes to override these default – and admittedly, rather useless – behaviours.

140c ⟨*Methods to be redefined by* Node*-derived classes* 140c⟩≡
```
METHOD AccumulateOnCharClosure (Code: INTEGER;
                                TheSet: NodeSet);
  BEGIN
  END AccumulateOnCharClosure;

METHOD EpsClosure (TheSet: NodeSet);
  BEGIN
  TheSet.Include (THIS);
  END EpsClosure;

METHOD AccumulateValidChars (TheCharSet: CharSet);
  BEGIN
  END AccumulateValidChars;
```

The `CharNode` class implements transitions on a single character, that must be provided at creation time.

CharNode

Node (See page 137)

TheCode holds the *Alphabet* (See page 13) symbol on which the node denoted by *self* can move to the inherited attribute *TheConnected*.

$TheCode : Alphabet$

TheCode is the only character for which there is a valid outgoing transition from *self*.

$ValidChars = \{TheCode\}$

OnCharClosure returns the ϵ-closure of the node reachable through *TheConnected* if given *TheCode*, and an empty set in all other cases.

$OnCharClosure(TheCode) = \delta\, TheConnected.\epsilon - closure$

$\forall\, c \in Alphabet \mid c \neq TheCode \bullet$

$\quad OnCharClosure(c).Data = \phi$

141 ⟨`Node`-*derived classes definition* 141⟩≡

```
CLASS CharNode;
  INHERITS Node;

  REDEFINE METHOD CREATE (TheCode: INTEGER);
END CharNode;
```

142 ⟨Node-*derived classes* 142⟩≡

```
CLASS CharNode;
  INHERITS Node;

  VAR
    TheCode: INTEGER;

  REDEFINE METHOD CREATE (TheCode: INTEGER);
    BEGIN
    BASE;
    THIS.TheCode ← TheCode;
    END CREATE;

  REDEFINE METHOD AccumulateOnCharClosure (Code: INTEGER;
                                           TheSet: NodeSet);
    BEGIN
    IF Code = TheCode THEN
      Connected.EpsClosure (TheSet);
      END;
    END AccumulateOnCharClosure;

  REDEFINE METHOD AccumulateValidChars (TheCharSet: CharSet);
    BEGIN
    TheCharSet.Include (TheCode);
    END AccumulateValidChars;

END CharNode;
```

`EpsNode` denotes a node which can be visited without consuming a character from the input. On the other hand, it also supports the ability to connect to a second outgoing `Node` in addition to the first outgoing `Node` all `Node`-derived classes support.

$\epsilon - Node$

$Node$ (See page 137)

$TheConnected_2 : \rho \downarrow Node$

$ValidChars$ is constrained to an empty set. There is no character acceptable in the $Node$ denoted by $self$, which does not imply that there is no such acceptable character in one of the $Node$s in $self$'s ϵ-closure.

$ValidChars = \phi$

$OnCharClosure$'s constraint stated here is consistent with the properties of its original declaration (See page 137) and $ValidChars$ as constrained above.

$\forall\, c \in Alphabet \bullet$
$\quad OnCharClosure(c).Data = \phi$

Well, this class must serve a useful purpose. $\epsilon - closure$, as constrainted here, includes the ϵ-closure of the two connected $Node$s. It is used to build graphs of $Node$s.

$TheConnected \neq \Lambda \Rightarrow$
$\qquad \delta\, TheConnected.\epsilon - closure.Data \subset \epsilon - closure.Data$
$TheConnected_2 \neq \Lambda \Rightarrow$
$\qquad \delta\, TheConnected_2.\epsilon - closure.Data \subset \epsilon - closure.Data$

143 ⟨`Node`-*derived classes definition* 141⟩+≡

```
CLASS EpsNode;
  INHERITS Node;

  REDEFINE METHOD CREATE;
  METHOD Connect2 (Other: Node);
END EpsNode;
```

144a ⟨Node-*derived classes* 142⟩+≡

```
CLASS EpsNode;
  INHERITS Node;

  VAR
    TheConnected2: Node;

  REDEFINE METHOD CREATE;
    BEGIN
    BASE;
    END CREATE;

  METHOD Connect2 (Other: Node);
    BEGIN
    TheConnected2 ← Other;
    END Connect2;

  ⟨EpsNode's EpsClosure method 144b⟩

END EpsNode;
```

The `EpsClosure` method as redefined by `EpsNode` class is the only non-trivial version of this method. Its purpose if to compute the set of all the nodes that can be reached through an ϵ-transition starting at the current node.

This method is *lazy* in the sense that the first time it is called, the ϵ-closure is computed, and stored in an attribute so that it can be returned for all subsequent calls of this method without having to be recomputed.

This is of course only made possible by the fact that these closures are computed when the automaton is not altered anymore, that is, when no more new nodes are added. Otherwise, from one invocation to another, the ϵ-closure might be different.

144b ⟨EpsNode*'s* EpsClosure *method* 144b⟩≡

```
VAR
  MyEpsClosure: NodeSet;
```

This method is recursive. Whenever a node is added, that node's ϵ-closure must be added to the set.

145 ⟨EpsNode*'s* EpsClosure *method* 144b⟩+≡

```
REDEFINE METHOD EpsClosure (TheSet: NodeSet);

  METHOD Visit(n1, n2: Node);
    VAR
      PleaseVisit1,
      PleaseVisit2: BOOLEAN;
    BEGIN
    IF n1 ≠ Λ THEN
      IF ¬ MyEpsClosure.IncludesNode (n1) THEN
        MyEpsClosure.Include (n1);
        PleaseVisit1 ← TRUE;
        END;
      END;
    IF n2 ≠ Λ THEN
      IF ¬ MyEpsClosure.IncludesNode (n2) THEN
        MyEpsClosure.Include (n2);
        PleaseVisit2 ← TRUE;
        END;
      END;
    IF PleaseVisit1 THEN
      n1.EpsClosure(MyEpsClosure);
      END;
    IF PleaseVisit2 THEN
      n2.EpsClosure(MyEpsClosure);
      END;
    END Visit;

  BEGIN
  IF MyEpsClosure = Λ THEN
    MyEpsClosure.CREATE;
    MyEpsClosure.Include (THIS);
    Visit (Connected, TheConnected2);
    END;
  TheSet.Or(MyEpsClosure);
  END EpsClosure;
```

`CharSetNode` is a generalization of `CharNode` (See page 141), where, instead of having a single character on which the transition can be triggered, we have a *set* of such characters. The distinction between the two is essentially historical: `CharSetNode` has been introduced long after `CharNode`, the distinction has been maintained for resource consumption reasons only, and a careful reimplementation might merge these two classes.

CharSetNode

$Node$ (See page 137)

$TheCharSet : \mathbb{P}\, Alphabet$ (See page 13)

$TheCharSet \neq \phi$
$ValidChars = TheCharSet$

$OnCharClosure$ is constrained to return the ϵ-closure of $TheConnected$ if its argument is one of the valid characters.

$\forall\, c \in TheCharSet \bullet$
$\quad OnCharClosure(c) = \delta\, TheConnected.\epsilon - closure$
$\forall\, c \in Alphabet \mid c \notin TheCharSet \bullet$
$\quad OnCharClosure(c).Data = \phi$

146a ⟨Node-*derived classes definition* 141⟩+≡

```
    CLASS CharSetNode;
      INHERITS Node;

      REDEFINE METHOD CREATE (TheCharSet: CharSet);
    END CharSetNode;
```

146b ⟨Node-*derived classes* 142⟩+≡

```
  CLASS CharSetNode;
    INHERITS Node;

    VAR
      TheCharSet: CharSet;

    REDEFINE METHOD CREATE (TheCharSet: CharSet);
      BEGIN
      BASE;
      THIS.TheCharSet ← TheCharSet;
      END CREATE;

    ⟨CharSetNode's AccumulateOnCharClosure 147a⟩
    ⟨CharSetNode's AccumulateValidChars 147b⟩

  END CharSetNode;
```

The set of nodes that can be reached starting at a `CharSetNode` on a `Ch` transition is the ϵ-closure of its connected node if `Ch` is in the set of characters attached to this node, and empty otherwise.

147a ⟨`CharSetNode`*'s* `AccumulateOnCharClosure` 147a⟩≡
```
  REDEFINE METHOD AccumulateOnCharClosure (Code: INTEGER;
                                          TheSet: NodeSet);
    BEGIN
    IF TheCharSet.Includes (Code) THEN
      Connected.EpsClosure (TheSet);
      END;
    END AccumulateOnCharClosure;
```

The characters for which a valid transition exists starting at a `CharSetNode` is defined by its attached set of characters. Hence, the attribute `CharSetNode` is or'ed with the formal `CharSetNode` in order to accumulate these characters in the formal.

147b ⟨`CharSetNode`*'s* `AccumulateValidChars` 147b⟩≡
```
  REDEFINE METHOD AccumulateValidChars (TheCharSet: CharSet);
    BEGIN
    TheCharSet.Or(THIS.TheCharSet);
    END AccumulateValidChars;
```

`AcceptNode` is a node that refers to a `Token` (See page 54), to indicate that when in a NFA state that contains this `Node`, the automaton has recognized a lexeme that matches the attached `Token`'s definition. By construction, the outgoing `Node` connection is always `VOID` (Λ).

AcceptNode

Node (See page 137)

$AcceptingToken : TokenCode$ (See page 14)

$\forall\, c \in Alphabet \bullet OnCharClosure(c).Data = \phi$
$ValidChars = \phi$

147c ⟨`Node`-*derived classes definition* 141⟩+≡
```
    CLASS AcceptNode;
      INHERITS Node;

      REDEFINE METHOD CREATE (TheToken: Token);
      METHOD AcceptingToken: Token;
    END AcceptNode;
```

There is not much to say about the implementation, of course.

148 ⟨Node-*derived classes* 142⟩+≡

```
CLASS AcceptNode;
  INHERITS Node;
  VAR
    TheToken: Token;

  REDEFINE METHOD CREATE (TheToken: Token);
    BEGIN
    BASE;
    THIS.TheToken ← TheToken;
    END CREATE;

  METHOD AcceptingToken: Token;
    BEGIN
    RESULT ← TheToken;
    END AcceptingToken;

END AcceptNode;
```

4.10 NodeSet

The module `NodeSet` describes sets of `Node`, sets of states of the NFA automaton. `NodeSet`'s are used when converting a NFA into a DFA. Each state of the DFA is a set of states (nodes) of the NFA.

⟨*Definition* ⟩≡

```
DEFINITION MODULE NodeSet;

IMPORT CharSet;
IMPORT HashElement;
FROM Node IMPORT Node;
IMPORT ArrayList;
FROM List IMPORT StringList;
FROM Streams IMPORT OutputStream;
FROM Token IMPORT Token;

  ⟨NodeSet class definition 151⟩
  ⟨NodeSetList class definition 170⟩

END NodeSet;
```

⟨*Implementation* ⟩≡

```
IMPLEMENTATION MODULE NodeSet;

FROM Node IMPORT NodeDictionary, AcceptNode;
IMPORT HashSpace;
IMPORT YLexCfg;
FROM DuraBitMap IMPORT DuraBitMap;
FROM Streams IMPORT StdOut;
IMPORT SYSTEM;
FROM Token IMPORT Token;

  ⟨NodeSet class implementation 152⟩

  CLASS NodeSetList;
    INHERITS ArrayList (NodeSet);

  ⟨NodeSetList class implementation 171a⟩
  END NodeSetList;

END NodeSet;
```

4.10.1 The `NodeSet` class

As such, `Nodes` (See page 137) are not very useful, since they are essentially states of a non-deterministic automaton. To reduce this non-determinism to a more efficient and manageable construction, we must work on *sets* of such nodes.

The purpose of the `NodeSet` class, as defined here, is to encapsulate such sets.

$NodeSet$

$Data : \mathbb{P}\ Node$ (See this page)

The ϵ-closure for a set of $Node$s is just the union of the ϵ-closures of the individual $Node$s in the set.

$\epsilon - closure : NodeSet$

$\epsilon - closure.Data = \bigcup_{x \in Data} x.\epsilon - closure.Data$

Similarly, the closure on a character $Code?$ from $Alphabet$ (See page 13) is the union of the closures on this same characters for all the $Node$s in the set.

$OnCharClosure : Alphabet \mapsto NodeSet$

$\forall\ a \in Alphabet \bullet$
$\quad OnCharClosure(a) = \bigcup_{x \in Data} x.OnCharClosure(a).Data$

$ValidTransitionsChars$ returns the set of all the characters for which a valid transition exists from at least a single $Node$ in the set.

$ValidTransitionsChars : \mathbb{P}\ Alphabet$

$ValidTransitionsChars = \{x \in Alphabet \mid OnCharClosure(x).Data \neq \phi\}$

$IsClosed$ indicates whether the set of $Node$s is ϵ-closed, that is, whether all the $Nodes$ that can be reached from a $Node$ in the set without consuming a character are in the set.

$IsClosed : \mathbb{B}$

$IsClosed =$ true $\Leftrightarrow$
$\quad Data = \epsilon - closure.Data$

151 ⟨`NodeSet` *class definition* 151⟩≡

```
CLASS NodeSet;
  INHERITS HashElement;

  REDEFINE METHOD CREATE;

  ⟨Inclusion management, definition 153b⟩
  ⟨NodeRow method definition 154b⟩
  ⟨NodeSet elementary methods, definition 155a⟩
  ⟨OnCharClosure method definition 157d⟩
  ⟨EpsClosure method definition 156b⟩
  ⟨ValidTransitionsChars method definition 158a⟩
  ⟨AcceptingRow method definition 159b⟩
  ⟨SameAccepting method definitions 160b⟩
  ⟨Connect method definition 161c⟩
  ⟨GenerateResultArray method definition 167b⟩
  ⟨GenerateCode method definition 162d⟩
  ⟨Numbering management, definition 162b⟩
END NodeSet;
```

The `NodeSet` class implements a list of `Node` instances, represented as a bitmap. Each node must be referenced in the `NodeDictionary` and its index in this dictionary is used for its corresponding bit in the bitmap.

```
152  ⟨NodeSet class implementation 152⟩≡
       CLASS NodeSet;
         INHERITS HashElement;

         ⟨The actual bitmap 153a⟩

         ⟨NodeSet connection management 161b⟩

         ⟨GenerateCode method implementation 163⟩
         ⟨GenerateResultArray method implementation 168⟩

         ⟨Numbering methods, implementation 162c⟩

         ⟨AcceptingTokens method implementation 159a⟩
         ⟨AcceptingRow method implementation 160a⟩
         ⟨SameAccepting method implementation 161a⟩

         REDEFINE METHOD CREATE;
           BEGIN
           BASE;
           TheBitMap.CREATE (NodeDictionary.High+1);
           TheHashValue ← -1;
           END CREATE;

         ⟨EpsClosure method implementation 157a⟩
         ⟨ValidTransitionsChars method implementation 158b⟩
         ⟨OnCharClosure method implementation 157e⟩

         ⟨Hash value management 162a⟩

         ⟨NodeSet elementary methods, implementation 155b⟩
         ⟨NodeRow method implementation 154c⟩
         ⟨Inclusion management, implementation 153c⟩

       END NodeSet;
```

`TheBitMap` holds a bitmap (See page 843) that indicates the nodes included in the `NodeSet`, using the numbering scheme attached to `Nodes` (See method *Node.Node.Nr* on page 139).

YAFL

This simple attribute has gone through several versions. First, `Set` class as provided by the YAFL standard library has been used, but it proved dramatically inefficient. This performance problem is easy to understand, since the sets as supported by the YAFL standard library are based on lists, memberships test being based on a linear search.

Then, it used `BitMaps` as provided by the YAFL standard library. They are simple and efficient, but actually allocate each bit, even if the bitmaps happen to be sparse in practice. When dealing with ever growing lexers, this simplistic scheme soon failed due to excessive random access memory requirements.

I then introduced an intermediate class named `LexBitMap`, which, depending on a configuration option, used plain `BitMaps` for speed, or `DuraBitMaps` (See page 843) to reduce the storage requirement by using a sparse bitmap mechanism. Amazingly enough, using these memory-optimized bitmaps was actually often *faster* than using the so-called performance optimized ones, because of the reduced pressure on memory, and consequently, the reduced frequency of mark and sweep process of the garbage collector.

All in all, I finally removed this intermediate class, and replaced it by a direct reference to `DuraBitMap`, improving performance by removing the extra level of indirection induced by `LexBitMap`. Consequently, the command-line options for memory- and speed- optimizations are removed as well. In the cases where the original speed-optimized scheme might have been more efficient, we are dealing with lexer that are small enough to make the performance of their generation irrelevant anyway.

One might also argue that it might have been better to detach `DuraBitMap` from `DURA` and store it in a library which is then shared by both `lexyt` and `DURA`. As a matter of methodology, it would have been far better. However, since the declaration of the attribute here under is the *unique* reference from `lexyt` to `DURA`, it did not seem to be worth the trouble to go though the definition and documentation of a library of reusable components.

153a ⟨*The actual bitmap* 153a⟩≡

```
VAR
  TheBitMap: DuraBitMap;
```

The `IncludesNode` method indicates whether the node `TheNode` is in the node set.

153b ⟨*Inclusion management, definition* 153b⟩≡

```
METHOD IncludesNode (TheNode: Node): BOOLEAN;
  POST
    NO_CHANGE;
```

153c ⟨*Inclusion management, implementation* 153c⟩≡

```
METHOD IncludesNode (TheNode: Node): BOOLEAN;
  BEGIN
  RESULT ← TheBitMap[TheNode.Nr];
  END IncludesNode;
```

The `Include` method marks a given node as being part of the node set.

153d ⟨*Inclusion management, definition* 153b⟩+≡

```
METHOD Include (TheNode: Node);
  POST
    OLD IncludesNode(TheNode) ⇒ NO_CHANGE,
    IncludesNode(TheNode);
```

It basically invalidates the lazily evaluated row and the cached hash key value, and sets the bit corresponding to the node TheNode in the attached bitmap.

154a ⟨*Inclusion management, implementation* 153c⟩+≡
```
METHOD Include (TheNode: Node);
  BEGIN
  RowValid ← FALSE;
  TheHashValue ← -1;
  TheBitMap.Set (TheNode.Nr);
  END Include;
```

NodeRow returns all the Nodes that are included into THIS in the form of an array. Its evaluation is lazy, controlled by the RowValid attribute. Whenever a new Node is added to THIS, this lazily evaluated row is invalidated so that any subsequent call to NodeRow recomputes a fresh array instead of repeatedly returning the same one.

154b ⟨NodeRow *method definition* 154b⟩≡
```
METHOD NodeRow: ARRAY OF Node;
```

154c ⟨NodeRow *method implementation* 154c⟩≡
```
VAR
  RowValid: BOOLEAN;
  TheRow: ARRAY OF Node;

METHOD NodeRow: ARRAY OF Node;
  VAR
    TempList: ArrayList(Node);
  BEGIN
  IF ¬ RowValid THEN
    TheRow ← Λ;
    RowValid ← TRUE;
    FOR i ∈ 0 .. Size-1 | Get(i) DO
      IF TempList = Λ THEN
        TempList.CREATE;
        END;
      TempList.Append (NodeDictionary_i;
      END;
    IF TempList ≠ Λ THEN
      TempList.Row ← TempList.Row;
      END;
    END;
  RESULT ← TheRow;
  END NodeRow;
```

155a ⟨`NodeSet` *elementary methods, definition* 155a⟩≡

```
METHOD Dump;
METHOD IsClosed: BOOLEAN;
METHOD IsEqual (Other: NodeSet): BOOLEAN;
METHOD Or (Other: NodeSet);
METHOD Size: INTEGER;
METHOD UsedBits: INTEGER;
METHOD UsedPages: INTEGER;
```

`Dump` simply displays the bits set in the bit map on the standard output stream.

155b ⟨`NodeSet` *elementary methods, implementation* 155b⟩≡

```
METHOD Dump;
  BEGIN
  FOR i ∈ 0 .. Size-1 | THIS_i DO
    StdOut.WriteInt (i,8);
    END;
  StdOut.WriteLn;
  END Dump;
```

155c ⟨`NodeSet` *elementary methods, implementation* 155b⟩+≡

The following methods simply delegate the call to the embedded `DuraBitMap` (See page 843).

156a ⟨`NodeSet` *elementary methods, implementation* 155b⟩+≡

```
METHOD Size: INTEGER;
  BEGIN
  RESULT ← TheBitMap.Size;
  END Size;

METHOD UsedBits: INTEGER;
  BEGIN
  RESULT ← TheBitMap.UsedBits;
  END UsedBits;

METHOD UsedPages: INTEGER;
  BEGIN
  RESULT ← TheBitMap.UsedPages;
  END UsedPages;

METHOD Get(Index: INTEGER): BOOLEAN;
  BEGIN
  RESULT ← TheBitMap_Index;
  END Get;

METHOD Set(Index: INTEGER);
  BEGIN
  TheBitMap.Set(Index);
  END Set;

METHOD IsEqual (Other: NodeSet): BOOLEAN;
  BEGIN
  RESULT ← TheBitMap.IsEqual(Other.TheBitMap);
  END IsEqual;

METHOD Or (Other: NodeSet);
  BEGIN
  TheBitMap.Or(Other.TheBitMap);
  END Or;
```

The `EpsClosure` method computes the ϵ-closure of the node set it is being applied on, by returning the set union of the ϵ-closure of all its elements.

156b ⟨`EpsClosure` *method definition* 156b⟩≡

```
METHOD EpsClosure: NodeSet;
  POST
    RESULT = RESULT.EpsClosure;
```

157a ⟨`EpsClosure` *method implementation* 157a⟩≡
```
VAR
  TheClosure: NodeSet;

METHOD EpsClosure: NodeSet;
  BEGIN
  IF TheClosure = Λ THEN
    TheClosure.CREATE;
    FOR p ∈ NodeRow DO
      ⟨Visit p 157b⟩
      END;
    END;
  RESULT ← TheClosure;
  END EpsClosure;
```

When visiting a given node, we first add it to the closure (every `Node` in `THIS` is member of `THIS`'s closure), and then accumulate the node's own closure into the `THIS`'s closure (See method *Node.Node.EpsClosure* on page 138).

157b ⟨*Visit p* 157b⟩≡
```
TheClosure.Include(p);
p.EpsClosure(TheClosure);
```

157c ⟨`EpsClosure` *method implementation* 157a⟩+≡
```
METHOD IsClosed: BOOLEAN;
  BEGIN
  RESULT ← EpsClosure.IsEqual (THIS);
  END IsClosed;
```

`OnCharClosure` can only be applied onto closed `NodeSet`s, and returns a `Node-Set` that contains the epsc of the `Node`'s one can reach starting at `THIS` on a transition on character `Code`.

157d ⟨`OnCharClosure` *method definition* 157d⟩≡
```
METHOD OnCharClosure(Code: INTEGER): NodeSet;
PRE
  IsClosed;
POST
  RESULT.IsClosed;
```

The implementation is rather simple: it relies on `AccumulateOnCharClosure` (See page 140) which provides the same service on a single `Node`.

157e ⟨`OnCharClosure` *method implementation* 157e⟩≡
```
METHOD OnCharClosure(Code: INTEGER): NodeSet;
  BEGIN
  RESULT.CREATE;
  FOR p ∈ NodeRow DO
    p.AccumulateOnCharClosure(Code, RESULT);
    END;
  END OnCharClosure;
```

The `ValidTransitionsChars` method returns a character set that contains all the characters for which a transition has been defined starting at the `NodeSet` denoted by `THIS`. It is computed by taking the union (in terms of sets of characters) of all nodes included into `THIS`, and computing the set of valid characters for each of them.

158a ⟨`ValidTransitionsChars` *method definition* 158a⟩≡

```
METHOD ValidTransitionsChars: CharSet;
  PRE
    IsClosed;
```

Most of `ValidTransitionChars`'s intelligence lies in its call to `AccumulateValidChars` (See page 138) that performs the same function, but on a single `Node` rather than on a set of `Node`s.

`ValidTransitionsChars` is also a lazy method, that computes its return value on the first invocation, and that keeps it in such a way that it can be returned without any further computation for any subsequent call.

158b ⟨`ValidTransitionsChars` *method implementation* 158b⟩≡

```
VAR
  TheValidTransitionsChars: CharSet;

METHOD ValidTransitionsChars: CharSet;
  BEGIN
  IF TheValidTransitionsChars = Λ THEN
    TheValidTransitionsChars.CREATE;
    FOR p ∈ NodeRow DO
      p.AccumulateValidChars(TheValidTransitionsChars);
      END;
    END;
  RESULT ← TheValidTransitionsChars;
  END ValidTransitionsChars;
```

The `AcceptingTokens` method returns the list of tokens that are accepted by the current state. In other words, it is the set of possible interpretations of the input when the lexical analyzer is in the DFA state referred to by `THIS`.

159a ⟨`AcceptingTokens` *method implementation* 159a⟩≡

```
METHOD AcceptingTokens: StringList(Token);
  POST
    NO_CHANGE;
  BEGIN
  FOR p ∈ NodeRow DO
    WHAT p OF
      IN AcceptNode:
        IF RESULT = Λ THEN
          RESULT.CREATE;
          END;
        RESULT.Append (TAG.AcceptingToken);
        END;
     ELSE
      -- Don't abort, because the set can contain
      -- Nodes of all kinds, not just AcceptNode's.
      END;
    END;
  END AcceptingTokens;
```

`AcceptingRow` returns the same information as `AcceptingToken` but structured as an array instead of a list. It also has the nice property of being *lazy* in the sense that the result that is returned in its first invocation on an instance is returned without further processing when the method is called again on that same instance.

159b ⟨`AcceptingRow` *method definition* 159b⟩≡

```
METHOD AcceptingRow: ARRAY OF Token;
```

AcceptingRow's laziness (as defined above) is controlled by AcceptingRowAsked which is TRUE if TheAcceptingRow holds the array of the tokens that are accepted when the DFA is in the state referred to by THIS.

160a ⟨AcceptingRow *method implementation* 160a⟩≡

```
VAR
  AcceptingRowAsked: BOOLEAN;
  TheAcceptingRow: ARRAY OF Token;

METHOD AcceptingRow: ARRAY OF Token;
VAR
  TokList: StringList(Token);
BEGIN
IF ¬ AcceptingRowAsked THEN
  AcceptingRowAsked ← TRUE;
  TokList ← AcceptingTokens;
  IF TokList ≠ Λ THEN
    TheAcceptingRow ← TokList.Row;
    END;
  END;
RESULT ← TheAcceptingRow;
END AcceptingRow;
```

The SameAccepting method indicates whether THIS and another NodeSet passed as parameter accept the same set of tokens. It is used in GenerateResultArrays (See page 172).

160b ⟨SameAccepting *method definitions* 160b⟩≡

```
METHOD SameAccepting (Other: NodeSet): BOOLEAN;
  POST
    NO_CHANGE;
```

It is a rather straightforward piece of code, based on the assumption that if the two rows contain the same elements, they will be ordered the same way. It is indeed the case, since AcceptingRow (See the preceding page) calls Accepting-Tokens (See the page before), which, in turn calls NodeRow (See page 154), which goes through all the elements in the dictionary (See page 135) in growing index order, and checks whether every Node is in the set or not. The order in which the various collections are built is thus deterministic.

Anyway, since this method is used to optimize the result arrays so that several states can share their result array, a malfunction of `SameAccepting` would not consequence beyond a possible increase of memory consumption.

161a ⟨`SameAccepting` *method implementation* 161a⟩≡

```
METHOD SameAccepting (Other: NodeSet): BOOLEAN;
  VAR
    MyRow, OtherRow: ARRAY OF Token;
  BEGIN
  MyRow ← AcceptingRow;
  OtherRow ← Other.AcceptingRow;
  IF (OtherRow = Λ) ∧ (MyRow = Λ) THEN
    RESULT ← TRUE;
   ELSIF (OtherRow ≠ Λ) ∧ (MyRow ≠ Λ) THEN
    IF OtherRow.SIZE = MyRow.SIZE THEN
      RESULT ← TRUE;
      FOR i ∈ 0 .. OtherRow.SIZE - 1 WHILE RESULT DO
        RESULT ← MyRow_i = OtherRow_i;
        END;
      END;
    END;
  END SameAccepting;
```

Each `NodeSet` instance can be connected to one other `NodeSet` instance for every input character that occurs. This transition ability is implemented by `Connected`, that is an array of `NodeSet`s where the i^{th} element indicates the `NodeSet` to move to when receiving code i.

161b ⟨`NodeSet` *connection management* 161b⟩≡

```
VAR
  Connected: ARRAY OF NodeSet;
```

`Connect` can be used to connect `THIS` to `Target` on the input `Code`.

161c ⟨`Connect` *method definition* 161c⟩≡

```
METHOD Connect (Code: INTEGER;
                Target: NodeSet);
```

`Connected` is allocated on demand.

161d ⟨`NodeSet` *connection management* 161b⟩+≡

```
METHOD Connect (Code: INTEGER;
                Target: NodeSet);
  BEGIN
  IF Connected = Λ THEN
    Connected.CREATE (YLexCfg.MaxTokenCode);
    END;
  Connected_Code ← Target;
  END Connect;
```

We make extensive use of the ability to store `NodeSets` in ad hoc hash tables, but we might suffer some performance problem if computing the hash keys is too costly. Computing the hash value of a `NodeSet` is based on the hash value as computed by bitmaps, and the so computed value is retained for instantaneous further computation. YAFL

162a ⟨*Hash value management* 162a⟩≡

```
VAR
  TheHashValue: INTEGER;

REDEFINE METHOD HashValue: INTEGER;
  BEGIN
  IF TheHashValue < 0 THEN
    TheHashValue ← TheBitMap.HashValue;
    IF TheHashValue < 0 THEN
      TheHashValue ← - TheHashValue;
      END;
    END;
  RESULT ← TheHashValue;
  END HashValue;
```

The following two methods set and read a number that can be attached to the `NodeSet`. This number will represent the state number in the DFA.

162b ⟨*Numbering management, definition* 162b⟩≡

```
METHOD SetNr (TheNr: INTEGER);
  POST
    Nr = TheNr;
METHOD Nr: INTEGER;
  POST
    NO_CHANGE;
```

162c ⟨*Numbering methods, implementation* 162c⟩≡

```
VAR
  TheNr: INTEGER;

METHOD SetNr (TheNr: INTEGER);
  BEGIN
  THIS.TheNr ← TheNr;
  END SetNr;

METHOD Nr: INTEGER;
  BEGIN
  RESULT ← TheNr;
  END Nr;
```

162d ⟨`GenerateCode` *method definition* 162d⟩≡

```
METHOD GenerateCode (Output: OutputStream);
```

The first thing to do when generating the code for a DFA state is to set things according to the fact that this state has been reached: what are the current candidates for the honour of being the recognized token, set a variable indicating that the current token length is whatever it has to be, etc...

Then, one must also consider where to move depending on the current character that has been read. There are of course several different strategies to generate transition code. The first, easy way, would be to list one case for each possible character, and indicating which action should be taken for each of these characters.

```
CASE Ch OF
  0: Error := TRUE; END;
  1: NewState := 87; END;
  2: NewState := 87; END;
  3: Error := TRUE; END;
  4: NewState := 112; END;
  ...
```

Well, this would of course not be very satisfactory, since, in an automaton of, let's say, 100 states, we would end up having about 25600 transitions, which is quite a high price to pay for such a limited level of functionality.

Hence, we reversed the problem. Instead of building one transition for every input character, we built one transition for every *target state* where we might move, and build a single CASE alternative for all the input characters that would make us move to that target state.

```
CASE Ch OF
  1, 2: NewState := 87; END;
  4: NewState := 112; END;
  ...
 ELSE
  Error := TRUE;
  END;
```

Furthermore, we optimize the code generation even further by using YAFL's ability to represent CASE ranges efficiently, without having to list the values of the range one by one, as it is the case in C.

163 ⟨GenerateCode *method implementation* 163⟩≡

```
  METHOD GenerateCode (Output: OutputStream);
    VAR
      TokList: StringList(Token);

    ⟨Code generation method for a single transition 165⟩

    BEGIN
    ⟨First, generate code to set whatever has to be set in this state 164a⟩
    ⟨Then, generate transition code 164b⟩
    END GenerateCode;
```

The code that this chunk generates will only be activated if the compile-time constant BACKTRACKLEVEL is 0. Otherwise, it will be removed from the resulting code. Constant folding and dead code elimination are used here as elegant substitutes for conditional compilation.

164a ⟨*First, generate code to set whatever has to be set in this state* 164a⟩≡

```
TokList ← AcceptingTokens;
IF (TokList ≠ Λ) THEN
  ASSERT TokList.Size > 0;
  Output.WriteLine (“IF BACKTRACKLEVEL = 0 THEN”);
  Output.WriteLine (“  TheCurrentLength := AnalyzedSize;”);
  Output.WriteLine (“  AtomicToken := ” +
                               TokList.First.SymbolName + ‘‘;”);
  Output.WriteLine (“  END;”);
  END;
```

164b ⟨*Then, generate transition code* 164b⟩≡

```
IF Connected = Λ THEN
  Output.WriteLine (“State := ErrorState;”);
 ELSE
  Output.WriteLine (“CASE Ch OF”);
  ⟨Generate code for each possible target state we can move to 164c⟩
  Output.WriteLine (“ELSE State := ErrorState; END;”);
  END;
```

164c ⟨*Generate code for each possible target state we can move to* 164c⟩≡

```
FOR i ∈ 0 .. Connected.SIZE - 1 | Connected_i ≠ Λ DO
  GenerateTransition (Output, Connected_i.Nr);
  END;
```

The code for a single transition has always the form of

```
1, 7, 10 TO 13:
  State := 86;
  END;
```

`GenerateTransition` calls `GenerateRanges` to list the CASE values efficiently, taking advantage of large ranges using the `TO` keyword, then generates the change of value of `State`.

165 ⟨*Code generation method for a single transition* 165⟩≡

```
METHOD GenerateTransition (Output: OutputStream;
                           TargetNr: INTEGER);

  ⟨GenerateRanges method implementation 166⟩

  BEGIN
  GenerateRanges (Output, TargetNr);
  Output.WriteLine(“:”);
  Output.WriteString (“  State := ”);
  Output.WriteInt (TargetNr, 0);
  Output.WriteLine (“; END;”);
  END GenerateTransition;
```

When generating the code for the various ranges, `GenerateRanges` scans all the connected `NodeSet`'s, and builds the largest possible contiguous intervals of input characters for which the target is `TargetNr`.

Whenever it cannot extend a contiguous range any further, It calls `EmitRange` to actually emit the range denoted by $\mathit{FirstRange}..\mathit{LastRange}$.

166 ⟨`GenerateRanges` *method implementation* 166⟩≡

```
METHOD GenerateRanges (Output: OutputStream;
                        TargetNr: INTEGER);
  VAR
    FirstRange, LastRange: INTEGER;
    FirstTime: BOOLEAN;

    ⟨Method to emit a single range FirstRange..LastRange 167a⟩

  BEGIN
  FirstTime ← TRUE;
  FirstRange ← -1;
  FOR i ∈ 0 .. Connected.SIZE - 1 DO
    IF (Connected_i ≠ Λ) ∧ (Connected_i.Nr = TargetNr) THEN
      IF FirstRange < 0 THEN
        FirstRange ← i;
        LastRange ← i;
       ELSE
        IF i = LastRange + 1 THEN
          LastRange ← i;
         ELSE
          EmitRange(Output);
          FirstRange ← i;
          LastRange ← i;
          END;
        END;
      Connected_i ← Λ;
      END;
    END;
  IF FirstRange ≥ 0 THEN
    EmitRange(Output);
    END;
  END GenerateRanges;
```

`EmitRange` considers whether $LastRange - FirstRange$ is larger than `Threshold`. If it is, then the shortcut notation `A TO B` is used to represent the range $A..B$. Otherwise, the values are listed one by one.

167a ⟨*Method to emit a single range* `FirstRange..LastRange` 167a⟩≡

```
METHOD EmitRange (Output: OutputStream);
  CONST
    Threshold = 4;
  BEGIN
  ASSERT LastRange ≥ FirstRange;
  IF ¬ FirstTime THEN        -- Check whether this range is to be
                               -- appended to other ranges...
    Output.WriteChar (‘‘,’’);
   ELSE
    FirstTime ← FALSE;
    END;
  IF LastRange - FirstRange ≤ Threshold THEN
    FOR i ∈ FirstRange .. LastRange DO
      Output.WriteInt (i, 0);
      IF i ≠ LastRange THEN
        Output.WriteChar (‘‘,’’);
        END;
      END;
   ELSE
    Output.WriteInt (FirstRange, 0);
    Output.WriteString (‘‘ TO ’’);
    Output.WriteInt (LastRange, 0);
    END;
  END EmitRange;
```

167b ⟨`GenerateResultArray` *method definition* 167b⟩≡

```
METHOD GenerateResultArray (Output: OutputStream);
```

`GenerateResultArray` generates the code that initializes `ResultsArray` in the generated lexer (See *Lexical analyzer skeleton.Lexical analyzer skeleton.ResultsArray* on page 217), by producing something of the kind of:

```
ResultsArray[4].CREATE(3);
ResultsArray[4][0] := IfKeyword;
ResultsArray[4][1] := -Identifier;
ResultsArray[4][2] := -Any;
```

when applied to a `NodeSet` with number 4.

168 ⟨`GenerateResultArray` *method implementation* 168⟩≡

```
  METHOD GenerateResultArray (Output: OutputStream);
    CONST
      MaxTokens = 512;
    VAR
      TokList: StringList(Token);
      Toks: ONCE ARRAY OF Token;
      Negate: ONCE ARRAY OF BOOLEAN;
      Count: INTEGER;

      METHOD Prefix;
      BEGIN
      Output.WriteString (``ResultsArray['');
      Output.WriteInt (TheNr, 0);
      Output.WriteString (``]'');
      END Prefix;

    BEGIN
    TokList ← AcceptingTokens;
    IF (TokList ≠ Λ) ∧ (TokList.Size > 0) THEN
      ⟨Build Toks, Negate and Count for TokList 169a⟩
      ⟨Allocate and initialize the entry 169b⟩
      END;
    END GenerateResultArray;
```

169a ⟨*Build* Toks, Negate *and* Count *for* TokList 169a⟩≡

```
Count ← 0;
IF Toks = Λ THEN
  Toks.CREATE (MaxTokens);
  Negate.CREATE (MaxTokens);
 ELSE
  FOR i ∈ 0 .. Toks.SIZE - 1 DO
    Toks_i ← Λ;
    END;
  FOR i ∈ 0 .. Toks.SIZE - 1 DO
    Negate_i ← FALSE;
    END;
  END;
FOR Tok ∈ TokList |
             ∀x ∈ 0 .. Count-1 • Toks_x ≠ Tok DO
  Toks_Count ← Tok;
  Count ← Count + 1;
  FOR OtherTok ∈ Tok.Exclusion |
          ∀x ∈ 0 .. Count-1 • Toks_x ≠ OtherTok DO
    Toks_Count ← OtherTok;
    Negate_Count ← TRUE;
    Count ← Count + 1;
    END;
  END;
```

169b ⟨*Allocate and initialize the entry* 169b⟩≡

```
Prefix;
Output.WriteString (“.CREATE(”);
Output.WriteInt (Count, 0);
Output.WriteLine (“);”);
FOR i ∈ 0 .. Count - 1 DO
  Prefix;
  Output.WriteString (“[”);
  Output.WriteInt (i, 0);
  Output.WriteString (“] := ”);
  IF Negate_i THEN
    Output.WriteString (“ - ”);
    END;
  Output.WriteLine (Toks_i.SymbolName + “;”);
  END;
```

4.10.2 The `NodeSetList` class

$NodeSetList$

$NodeSetList$ is essentially a list of $NodeSet$s (See page 150) such that two elements in the list cannot have the same set of $Node$s (See page 137).

$$Data : \mathrm{seq}\ NodeSet$$

$$\forall\, a, b \in \mathrm{dom}\, Data : a \neq b \Leftrightarrow Data(a).Data \neq Data(b).Data$$

170 ⟨`NodeSetList` *class definition* 170⟩≡

```
CLASS NodeSetList;
  INHERITS ArrayList (NodeSet);

  REDEFINE METHOD CREATE;
  ⟨Includes method definition 172b⟩
  ⟨GenerateResultArrays method definition 172d⟩

  ⟨AutomatonArray method definition 174a⟩

  METHOD Renumber;

END NodeSetList;
```

171a ⟨`NodeSetList` *class implementation* 171a⟩≡

```
⟨NodeSetList attributes 171b⟩

METHOD Renumber;
  BEGIN
  FOR i ∈ 0 .. Size-1 DO
    Get(i).SetNr (i);
    END;
  END Renumber;

REDEFINE METHOD CREATE;
  BEGIN
  BASE;
  HashTable.CREATE(NodeDictionary.High * 4);
  END CREATE;

⟨Redefinition of the inherited Append method 171c⟩
⟨Includes method implementation 172c⟩
⟨GenerateResultArrays method implementation 173a⟩
⟨AutomatonArray method implementation 174b⟩
```

`HashTable` is an `HashSpace` that keeps track of all `Node`'s that have been stored in `THIS`, so that the `Includes` method can be implemented efficiently.

171b ⟨`NodeSetList` *attributes* 171b⟩≡

```
VAR
  HashTable: HashSpace (NodeSet);
```

171c ⟨*Redefinition of the inherited* `Append` *method* 171c⟩≡

```
REDEFINE METHOD Append(El: NodeSet);
  VAR
    Flag: BOOLEAN;
  BEGIN
  ASSERT Includes(El) = Λ;
  BASE(El);
  ⟨Actually store the element 172a⟩
  END Append;
```

When looking at this chunk, one might be tempted to believe that it might as well be written as: YAFL

```
ASSERT HashTable.Store(El);
```

However, such an apparent shortcut would cause the entire statement (hence, the storage of the element) to be discarded if the YAFL compile-time option to remove assertions is enabled. When assertions are discarded, they are ignored entirely, even if they execution include evaluation of expressions that might imply side effects.

172a ⟨*Actually store the element* 172a⟩≡

```
Flag ← HashTable.Store (El);
ASSERT Flag;
```

Given a `NodeSet` instance, the `Includes` method searches whether it already contains an equivalent `NodeSet`, in which case the already existing instance is returned. Two `NodeSet` instances are said to be equivalent when they contain the same `Node`'s.

Early profiling showed that this search took over 90% of the total processing time if implemented naively, using a sequential search. This is why it has been replaced by this optimized scheme based on hashing.

172b ⟨`Includes` *method definition* 172b⟩≡

```
METHOD Includes(n: NodeSet): NodeSet;
```

172c ⟨`Includes` *method implementation* 172c⟩≡

```
METHOD Includes(n: NodeSet): NodeSet;
  VAR
    Value: INTEGER;
    Candidate: NodeSet;
  BEGIN
  Value ← n.HashValue;
  Candidate ← HashTable.Find(Value);
  WHILE (Candidate ≠ Λ) ∧ (RESULT = Λ) DO
    IF Candidate.IsEqual(n) THEN
      RESULT ← Candidate;
     ELSE
      Candidate ← HashTable.Next;
      END;
    END;
  END Includes;
```

172d ⟨`GenerateResultArrays` *method definition* 172d⟩≡

```
METHOD GenerateResultArrays(Output: OutputStream);
```

173a ⟨`GenerateResultArrays` *method implementation* 173a⟩≡

```
METHOD GenerateResultArrays(Output: OutputStream);
  VAR
    CurNodeSet: NodeSet;
    j: INTEGER;
  BEGIN
  Output.WriteString (``ResultsArray.CREATE(");
  Output.WriteInt (Size+1, 0);
  Output.WriteLine (``);");
  FOR i ∈ 0 .. Size - 1 DO
    CurNodeSet ← THIS_i;
    j ← FIRST x ∈ 0 .. i−1 •
          CurNodeSet.SameAccepting(THIS_x;
    IF (j ≥ 0) ∧ (j < i) THEN
      ⟨Generate code if state i and state j have the same results 173b⟩
     ELSE
      CurNodeSet.GenerateResultArray (Output);
      END;
    END;
  END GenerateResultArrays;
```

A naive survey of this document might lead the reader into believing that no two em different states should have the same `AcceptingRow`. However, a small example can show how this can happen. For instance, let's consider a language with identifiers (made of one or more alphabetical characters) and two reserved words "`IF`" and "`DO`". We must have a state after the "`I`" and before the "`F`", and similarly, a state after the "`D`" and before the "`O`". These two intermediate states must be distinct. Otherwise, we would not be able to distinguish the reserved word "`DO`" from an identifier "`DF`". However, these two intermediate states yield the same `AcceptingRow`, indicating that when having reached these states, the lexer will have recognized an identifier so far.

173b ⟨*Generate code if state* `i` *and state* `j` *have the same results* 173b⟩≡

```
IF (CurNodeSet.AcceptingRow ≠ Λ) ∧
    (CurNodeSet.AcceptingRow.SIZE > 0) THEN
  Output.WriteString (``ResultsArray[");
  Output.WriteInt (i, 0);
  Output.WriteString (``] := ResultsArray[");
  Output.WriteInt (j, 0);
  Output.WriteLine (``];");
  END;
```

`AutomatonArray` is meant to return the transition function of the automaton as two dimension array of integers. The purpose of this construction is to be able to save this array onto a file that the lexer will read and interpret at runtime, rather than hard-wire nested `CASE` statements in the generated lexer. Such lexers, with externally defined transitions tables would be much smaller, hence much faster to compile, at the expense of some runtime performance penalty.

174a ⟨`AutomatonArray` *method definition* 174a⟩≡

```
METHOD AutomatonArray: ARRAY OF ARRAY OF INTEGER;
```

Well, it's not complete yet !!!

174b ⟨`AutomatonArray` *method implementation* 174b⟩≡

```
METHOD AutomatonArray: ARRAY OF ARRAY OF INTEGER;
  BEGIN
  ASSERT FALSE;
  FOR i ∈ 0 .. Size-1 DO
    -- CurNodeSet := Get(i);
    END;
  END AutomatonArray;
```

4.11 YLexCfg

The YLexCfg module is very simple and straightforward. It serves as central repository for the global configuration information that can be accessed from virtually any module of lexyt.

⟨*Definition* ⟩≡

```
DEFINITION MODULE YLexCfg;
  ONCE CLASS YLexCfg;

    REDEFINE METHOD CREATE;

    ⟨SetOption method definition 178⟩

    METHOD Help: BOOLEAN;
    METHOD Verbose: BOOLEAN;

    METHOD BackTrackLevel: INTEGER;
    METHOD SetBackTrackLevel (Level: INTEGER);
    METHOD DynamicBackTrackLevel: BOOLEAN;
    METHOD SetDynamicBackTrackLevel (Val: BOOLEAN);

    METHOD IgnoreCase: BOOLEAN;
    METHOD SetIgnoreCase (Val: BOOLEAN);

    METHOD UseAncillaryMethods: BOOLEAN;

    METHOD WordLength: INTEGER;
    METHOD GenerateYaccFName: ARRAY OF CHAR;

    METHOD MaxTokenCode: INTEGER;

    METHOD Interpreted: BOOLEAN;
    METHOD InterpretedTabFile: ARRAY OF CHAR;

    ⟨Error handling, definition 180d⟩

  END YLexCfg;

END YLexCfg;
```

⟨*Implementation* ⟩≡

```
IMPLEMENTATION MODULE YLexCfg;

FROM Conversions IMPORT IntConversions;
IMPORT SYSTEM;
FROM Streams IMPORT StdErr;

  ONCE CLASS YLexCfg;
    VAR
      TheHelp,
      TheVerbose,
      TheIgnoreCase,
      TheUseAncillaryMethods,
      TheDynamicBackTrackLevel: BOOLEAN;
      TheInterpretedTabFile,
      TheGenerateYaccFName: ARRAY OF CHAR;
      TheWordLength,
      TheMaxTokenCode,
      TheBackTrackLevel: INTEGER;

    REDEFINE METHOD CREATE;
      BEGIN
      TheWordLength ← 8;
      TheMaxTokenCode ← 256;
      BASE;
      END CREATE;

  ⟨Error handling, implementation 181⟩
  ⟨SetOption method implementation 179a⟩
```

⟨*Implementation* ⟩+≡
```
  METHOD Help: BOOLEAN;
    BEGIN
    RESULT ← TheHelp;
    END Help;

  METHOD Verbose: BOOLEAN;
    BEGIN
    RESULT ← TheVerbose;
    END Verbose;

  METHOD BackTrackLevel: INTEGER;
    BEGIN
    RESULT ← TheBackTrackLevel;
    END BackTrackLevel;

  METHOD SetBackTrackLevel (Level: INTEGER);
    BEGIN
    TheBackTrackLevel ← Level;
    END SetBackTrackLevel;
```

⟨*Implementation* ⟩+≡
```
  METHOD DynamicBackTrackLevel: BOOLEAN;
    BEGIN
    RESULT ← TheDynamicBackTrackLevel;
    END DynamicBackTrackLevel;

  METHOD SetDynamicBackTrackLevel (Val: BOOLEAN);
    BEGIN
    TheDynamicBackTrackLevel ← Val;
    END SetDynamicBackTrackLevel;

  METHOD IgnoreCase: BOOLEAN;
    BEGIN
    RESULT ← TheIgnoreCase;
    END IgnoreCase;

  METHOD SetIgnoreCase (Val: BOOLEAN);
    BEGIN
    TheIgnoreCase ← Val;
    END SetIgnoreCase;
```

⟨*Implementation* ⟩+≡

```
METHOD WordLength: INTEGER;
  BEGIN
  RESULT ← TheWordLength;
  END WordLength;

METHOD MaxTokenCode: INTEGER;
  BEGIN
  RESULT ← TheMaxTokenCode;
  END MaxTokenCode;

METHOD UseAncillaryMethods: BOOLEAN;
  BEGIN
  RESULT ← TheUseAncillaryMethods;
  END UseAncillaryMethods;

METHOD Interpreted: BOOLEAN;
  BEGIN
  RESULT ← TheInterpretedTabFile ≠ Λ;
  END Interpreted;

METHOD InterpretedTabFile: ARRAY OF CHAR;
  BEGIN
  RESULT ← TheInterpretedTabFile;
  END InterpretedTabFile;
```

⟨*Implementation* ⟩+≡

```
    METHOD GenerateYaccFName: ARRAY OF CHAR;
      BEGIN
      RESULT ← TheGenerateYaccFName;
      END GenerateYaccFName;

  END YLexCfg;

END YLexCfg;
```

The `SetOption` method recognizes command line option and sets the internal state of the `YLexCfg` ONCE class accordingly.

178 ⟨`SetOption` *method definition* 178⟩≡

```
METHOD SetOption (Opt: ARRAY OF CHAR);
```

179a ⟨SetOption *method implementation* 179a⟩≡

```
METHOD SetOption (Opt: ARRAY OF CHAR);
  VAR
    Err: BOOLEAN;
  BEGIN
  ASSERT Opt.SIZE > 0;
  CASE SYSTEM.UCASE(Opt_0) OF
    ⟨Depending on the code, take appropriate action 179b⟩
   ELSE
    Err ← TRUE;
    END;
  IF Err THEN
    StdErr.WriteLine (“Invalid option: ” + Opt);
    SetError (TRUE);
    END;
  END SetOption;
```

179b ⟨*Depending on the code, take appropriate action* 179b⟩≡

```
“H”:
  TheHelp ← TRUE;
  END;
“K”:
  TheUseAncillaryMethods ← TRUE;
  END;
```

The backtracking level can take any value from 0 to 3, or a character “D” to indicate that the backtracking level is dynamic, that is, it can be set at runtime.

179c ⟨*Depending on the code, take appropriate action* 179b⟩+≡

```
“B”:
  IF Opt.SIZE > 1 THEN
    CASE Opt_1 OF
      “0” .. “3”:
        SetBackTrackLevel (SYSTEM.ORD(Opt_1) - SYSTEM.ORD(“0”));
        END;
      “D”:
        SetDynamicBackTrackLevel (TRUE);
        END;
     ELSE
      Err ← TRUE;
      END;
   ELSE
    SetBackTrackLevel (0);
    END;
  END;
```

180a ⟨*Depending on the code, take appropriate action* 179b⟩+≡
```
“W”:
  IF Opt.SIZE > 1 THEN
    CASE Opt₁ OF
      “8”, “4”, “2”, “1”:
        Err ← FALSE;
        TheWordLength ← SYSTEM.ORD(Opt₁) - SYSTEM.ORD(“0”);
        END;
     ELSE
      Err ← TRUE;
      END;
    END;
  END;
“V”:
  TheVerbose ← TRUE;
  END;
“I”:
  TheIgnoreCase ← TRUE;
  END;
```

180b ⟨*Depending on the code, take appropriate action* 179b⟩+≡
```
“Y”:
  TheGenerateYaccFName ← Opt.SLICE (1, Opt.SIZE - 1);
  END;
“T”:
  TheMaxTokenCode ← IntConversions.StringToInt(
                                 Opt.SLICE (1, Opt.SIZE - 1));
  END;
```

The possibility of producing an interpreted lexer where the transitions are defined by tables of integers instead of a compiled lexer, where the transitions are represented by YAFL CASE statements has been investigated. This extension is justified by the hardly controlled growth of the lexers lexyt produces, up to a point where we might accept a slight performance penalty induced by the use of interpreted transition tables, for the sake of shorter compile times of the generated lexers *without* the transition logic. This feature is not entirely implemented yet, but an ad hoc command-line option is already supported.

180c ⟨*Depending on the code, take appropriate action* 179b⟩+≡
```
“F”:
  TheInterpretedTabFile ← Opt.SLICE (1, Opt.SIZE - 1);
  END;
```

180d ⟨*Error handling, definition* 180d⟩≡
```
METHOD SetError (Flag: BOOLEAN);
METHOD Error: BOOLEAN;
METHOD SetWarning (Flag: BOOLEAN);
METHOD Warning: BOOLEAN;
```

The following methods keep a global state information indicating whether errors or warnings have taken place.

181 ⟨*Error handling, implementation* 181⟩≡

```
VAR
  TheError,
  TheWarning: BOOLEAN;

METHOD SetError (Flag: BOOLEAN);
BEGIN
TheError ← Flag;
END SetError;

METHOD Error: BOOLEAN;
BEGIN
RESULT ← TheError;
END Error;

METHOD SetWarning (Flag: BOOLEAN);
BEGIN
TheWarning ← Flag;
END SetWarning;

METHOD Warning: BOOLEAN;
BEGIN
RESULT ← TheWarning;
END Warning;
```

4.12 Libraries

This section describes the `YLexicalAnalyzer` module where various classes are described. This module is not really part of the lexical analyzer generator. It contains utility classes that are referred to by all generated lexical analyzers.

4.13 YLexicalAnalyzer

⟨*Definition* ⟩≡

```
DEFINITION MODULE YLexicalAnalyzer;

FROM ArrayList IMPORT ArrayList;
FROM Comparable IMPORT Comparable;
FROM Streams IMPORT InputStream;

  ⟨YLexicalAnalyzer class definition 184a⟩
  ⟨AbstractCharStream class definition 191a⟩
  ⟨StringCharStream class definition 193c⟩
  ⟨RandomFileCharStream class definition 196⟩
  ⟨Comment class definition 206b⟩

END YLexicalAnalyzer;
```

⟨*Implementation* ⟩≡

```
IMPLEMENTATION MODULE YLexicalAnalyzer;

IMPORT SYSTEM;
FROM Streams IMPORT StdErr, StdOut;

  ⟨YLexicalAnalyzer class implementation 184b⟩
  ⟨AbstractCharStream class implementation 192a⟩
  ⟨StringCharStream class implementation 193d⟩
  ⟨RandomFileCharStream class implementation 197a⟩
  ⟨Comment class implementation 207a⟩

END YLexicalAnalyzer;
```

4.13.1 YLexicalAnalyzer

`YLexicalAnalyzer` is the base class for all the parsers generated by `lexyt`. It includes generic services, some deferred, some implemented and some implemented with possible redefinition in mind so that the default behaviour that can be used as is or extended through calls to `BASE`, or can be overruled by a more application-specific behaviour.

Table 4.2: The codes returned by the `YLexicalAnalyzer` class

19	`EOF`
20	`ERROR`
21	`SKIP`
32..8191	Meaningful user-defined symbols, such as identifiers, reserved words, punctuation
$8192..\infty$	Skippable user-defined symbols, such as white spaces and comments

184a ⟨`YLexicalAnalyzer` *class definition* 184a⟩≡

```
CLASS YLexicalAnalyzer;
  CONST
    ⟨Lexical analyzer's constants 184c⟩

    ⟨AttachFile method definition 186d⟩
    ⟨AttachCharStream method definition 185g⟩
    ⟨GetCharStream method definition 185i⟩
    ⟨LineCount method definition 186b⟩
    ⟨Skip method definition 188e⟩
    ⟨Comment handling, definition 189b⟩

    ⟨Deferred methods that are redefined by actual lexical analyzers 186f⟩

END YLexicalAnalyzer;
```

184b ⟨`YLexicalAnalyzer` *class implementation* 184b⟩≡

```
CLASS YLexicalAnalyzer;

  VAR
    ⟨YLexicalAnalyzer attributes 185f⟩

   ⟨AttachFile method implementation 186e⟩
   ⟨AttachCharStream method implementation 185h⟩
   ⟨GetCharStream method implementation 186a⟩
   ⟨LineCount method implementation 186c⟩
   ⟨Skip default method implementation 189a⟩
   ⟨Comment handling, implementation 189c⟩

END YLexicalAnalyzer;
```

`EofChar` is the constant that must be returned by the `Read` method (See page 191) to indicate that the end of the input has been reached.

184c ⟨*Lexical analyzer's constants* 184c⟩≡

```
EofChar = -1;
```

`EofChar` should not be confused with `EOF`, which is the constant returned by the *lexical analyzer* to indicate that the end of the file has been reached. Its actual value (19) is of no practical importance, as long as it is dictinct from the code of user-defined tokens (See table 4.2).

185a ⟨*Lexical analyzer's constants* 184c⟩+≡
```
EOF = 19;
```

`ERROR` is the code returned by the lexical analyzer if an unrecognizable token is found on the input or if the token found has been explicitely marked as being an error.

185b ⟨*Lexical analyzer's constants* 184c⟩+≡
```
ERROR = 20;
```

`SKIP` is the code used by the lexical analyzer if a token specifically marked as being skippable has been specified on the input. This code is merely used for the analyzer's internals, since such tokens should never be returned to the calling program.

185c ⟨*Lexical analyzer's constants* 184c⟩+≡
```
SKIP = 21;
```

`FIRSTTOKEN` is the reference constant for all the user-defined tokens, that will be defined as holding value `FIRSTTOKEN+1`, `FIRSTTOKEN+2`, etc...

185d ⟨*Lexical analyzer's constants* 184c⟩+≡
```
FIRSTTOKEN = 32;
```

`SKIPOFFSET` is the constant that will be added to the code for all tokens than can be skipped, such as space and comments. Its value must be large enough for the interval 32..[SKIPOFFSET-1]] to be able to hold the values of all meaningful user-defined tokens.

185e ⟨*Lexical analyzer's constants* 184c⟩+≡
```
SKIPOFFSET = 8192;
```

`TheCharStream` denotes the character stream (See class *YLexicalAnalyzer.AbstractCharStream* on page 190) where characters will be fetched.

185f ⟨`YLexicalAnalyzer` *attributes* 185f⟩≡
```
TheCharStream: AbstractCharStream;
```

`AttachCharStream` and `GetCharStream` resp. attach and consult the `CharStream` where `THIS` will fetch characters.

185g ⟨`AttachCharStream` *method definition* 185g⟩≡
```
METHOD AttachCharStream (CharStream: AbstractCharStream);
```

185h ⟨`AttachCharStream` *method implementation* 185h⟩≡
```
METHOD AttachCharStream (CharStream: AbstractCharStream);
  BEGIN
  TheCharStream ← CharStream;
  END AttachCharStream;
```

185i ⟨`GetCharStream` *method definition* 185i⟩≡
```
METHOD GetCharStream: AbstractCharStream;
```

186a ⟨`GetCharStream` *method implementation* 186a⟩≡

```
METHOD GetCharStream: AbstractCharStream;
  BEGIN
  RESULT ← TheCharStream;
  END GetCharStream;
```

`LineCount` serves as conveniency method, that computes the line count of a lexical analyzer by forwarding the request to the attached `CharStream` (See method*YLexicalAnalyzer.StringCharStream.LineCount* on page 195 and*YLexicalAnalyzer.RandomFileCharStream.LineCount* on page 200).

186b ⟨`LineCount` *method definition* 186b⟩≡

```
METHOD LineCount: INTEGER;
```

186c ⟨`LineCount` *method implementation* 186c⟩≡

```
METHOD LineCount: INTEGER;
  BEGIN
  RESULT ← TheCharStream.LineCount;
  END LineCount;
```

`AttachFile` is just a conveniency method to ease access to `AttachCharStream` when dealing with named files, rather than string-based char streams (See class *YLexicalAnalyzer.StringCharStream* on page 193). It creates a `RandomFileCharStream` (See page 195), opens it using `FName`, and attaches it to `THIS`.

186d ⟨`AttachFile` *method definition* 186d⟩≡

```
METHOD AttachFile (FName: ARRAY OF CHAR): BOOLEAN;
```

186e ⟨`AttachFile` *method implementation* 186e⟩≡

```
METHOD AttachFile (FName: ARRAY OF CHAR): BOOLEAN;
  VAR
    ChStream: RandomFileCharStream;
    Input: InputStream;
  BEGIN
  Input.CREATE;
  Input.Open (FName, Input.ReadAccess);
  IF Input.ErrorCode = Input.NoError THEN
    ChStream.CREATE (Input);
    AttachCharStream (ChStream);
    Input.Close;
    RESULT ← TRUE;
    END;
  END AttachFile;
```

`Advance` reads the *next* token from the input stream, and is redefined in the skeleton (See page 223). The resulting code can be fetched using the `CurrentToken` method.

186f ⟨*Deferred methods that are redefined by actual lexical analyzers* 186f⟩≡

```
DEFERRED METHOD Advance;
```

`ReadCurrentToken` reads entire token information at the current position in the input stream. It does not change this position. `ReadCurrentToken` is redefined in the skeleton (See page 219). The resulting code can be fetched using the `CurrentToken` method.

187a ⟨*Deferred methods that are redefined by actual lexical analyzers* 186f⟩+≡
```
DEFERRED METHOD ReadCurrentToken;
```

The code returned by the `CurrentToken` method is always a positive integer, that can be `EOF`, `ERROR` or a user-defined code. This method does not alter the lexical analyzer's internal state. `CurrentToken` is redefined in the skeleton (See page 217).

187b ⟨*Deferred methods that are redefined by actual lexical analyzers* 186f⟩+≡
```
DEFERRED METHOD CurrentToken: INTEGER;
-- POST
-- NO_CHANGE;
```

In backtracking lexical analyzers, `BackTrack` tries the next valid interpretation of the current token *without advancing the input pointer*. If no such alternate interpretation is available, or if the lexical analyzer has been generated without backtracking, `CurrentToken` returns `ERROR`. `BackTrack` is redefined in the skeleton (See page 224).

187c ⟨*Deferred methods that are redefined by actual lexical analyzers* 186f⟩+≡
```
DEFERRED METHOD BackTrack;
```

`SetBackTrackLevel` is meant to be called to set the backtrack level of lexers that have been marked as having a dynamic backtracking level. This method has no effect for lexers that have a statically defined backtrack level.

187d ⟨*Deferred methods that are redefined by actual lexical analyzers* 186f⟩+≡
```
DEFERRED METHOD SetBackTrackLevel (Level: INTEGER);
```

The `Image` method returns the *image* of the current token structured as a character string. `Image` is redefined in the skeleton (See page 228).

187e ⟨*Deferred methods that are redefined by actual lexical analyzers* 186f⟩+≡
```
DEFERRED METHOD Image: ARRAY OF CHAR;
```

The `TokenImage` method returns a character string that describes the current token. For instance, if the current token has been marked as not requiring that its image is kept, this method will return the token's name in textual form. Given a definition such as

```
LParen        \(
```

`TokenImage` will return `"LParen"` if `TokenNr` holds the value that represents `LParen`. This method is mainly used for debugging purposes. `TokenImage` is redefined in the skeleton (See page 222).

187f ⟨*Deferred methods that are redefined by actual lexical analyzers* 186f⟩+≡
```
DEFERRED METHOD TokenImage (TokenNr: INTEGER): ARRAY OF CHAR;
```

The next methods consult and reset the current position in the input stream, the current length, and the current incarnation (that is, the current interpretation of the input for the current length). For backtracking purposes, these values cannot only be read, they can also be reset.

188a ⟨*Deferred methods that are redefined by actual lexical analyzers* 186f⟩+≡
```
DEFERRED METHOD CurrentPosition: INTEGER;
DEFERRED METHOD CurrentLength: INTEGER;
DEFERRED METHOD CurrentIncarnation: INTEGER;

DEFERRED METHOD SetCurrentPosition (Pos: INTEGER);
DEFERRED METHOD SetCurrentLength (Len: INTEGER);
DEFERRED METHOD SetCurrentIncarnation (Carn: INTEGER);
```

`ReSync` is just a conveniency method that encapsulates calls to `SetCurrentPosition`, `SetCurrentLength` and `SetCurrentIncarnation`.

188b ⟨*Deferred methods that are redefined by actual lexical analyzers* 186f⟩+≡
```
DEFERRED METHOD ReSync (Pos, Len, Carn: INTEGER);
```

After backtracking several times, or after using the `SetCurrentLength` method, one can reset the lexer to consider the maximum length lexeme at the current position, which is the length considered just after that `ReadCurrentToken` is read.

188c ⟨*Deferred methods that are redefined by actual lexical analyzers* 186f⟩+≡
```
DEFERRED METHOD ResetMaximumLength;
```

`LastToken` returns a code that indicates the highest code for lexemes returned by a lexical analyzer. Lexers generated by `lexyt` produce a constant denoted `LASTTOKEN` but that cannot be used in a polymorphic way, because YAFL does not support late binding for constants. Accessing a constant must result in a compile-time evaluation.

188d ⟨*Deferred methods that are redefined by actual lexical analyzers* 186f⟩+≡
```
DEFERRED METHOD LastToken: INTEGER;
```

`Skip` is called by the lexical analyser when skipping a token (for instance, a comment) so that the very event of this skip can be captured.

188e ⟨`Skip` *method definition* 188e⟩≡
```
METHOD Skip(Code: INTEGER);
```

The default behaviour for the `Skip` is to do consider those of the comments that must be retained, and to store the comment if necessary.

189a ⟨`Skip` *default method implementation* 189a⟩≡

```
VAR
  TheKeepComments: ARRAY OF INTEGER;
  TheComments: ArrayList(Comment);

METHOD Skip(Code: INTEGER);
  VAR
    AComment: Comment;
  BEGIN
  IF ∃ x ∈ TheKeepComments • x = Code THEN
    AComment.CREATE (Code, CurrentPosition, CurrentLength);
    IF TheComments = Λ THEN
      TheComments.CREATE;
      END;
    TheComments.Append (AComment);
    END;
  END Skip;
```

189b ⟨*Comment handling, definition* 189b⟩≡

```
METHOD SetKeepComments (TheArr: ARRAY OF INTEGER);
METHOD GetComments: ArrayList(Comment);
METHOD ClearComments;
METHOD AjustCommentLineAndColNr;
```

`SetKeepComments` can be used to indicate which of the skippable tokens in a lexical specification ought to be retained as comment, as opposed to irrelevant tokens such as white space or line breaks.

189c ⟨*Comment handling, implementation* 189c⟩≡

```
METHOD SetKeepComments (TheArr: ARRAY OF INTEGER);
  BEGIN
  TheKeepComments ← TheArr;
  END SetKeepComments;
```

189d ⟨*Comment handling, implementation* 189c⟩+≡

```
METHOD GetComments: ArrayList(Comment);
  BEGIN
  RESULT ← TheComments;
  END GetComments;

METHOD ClearComments;
  BEGIN
  TheComments ← Λ;
  END ClearComments;
```

AjustCommentLineAndColNr calls ComputeLineAndColNr (See page 210) to translate character offset positions into line and column numbers, admittedly far easier to handle.

190 ⟨*Comment handling, implementation* 189c⟩+≡

```
METHOD AjustCommentLineAndColNr;
  BEGIN
  FOR Comment ∈ TheComments DO
    Comment.ComputeLineAndColNr (TheCharStream);
    END;
  END AjustCommentLineAndColNr;
```

4.13.2 AbstractCharStream base class

AbstractCharStream incarnates the char stream that can be lexed, that is, separated into lexemes. It is a base class that must be inherited in order to provide streams that analyze a file or a character string, or any similar character flow.

However, it is worth noticing that in practice, they will probably all end up being quite similar to the string case, since lexyt supports backtracking. Hence, it must be able to resynchronize on any past token, and the simplest way to achieve this is by reading the entire input in memory.

The only exception one can imagine would be an AbstractCharStream-derived class that would swap parts of an input in and out to be able to perform lexical analysis on huge (Typically, $> 100\,MB$) inputs, so huge that reading them all in memory is unpractical.

This module includes two derivations of `AbstractCharStream`:

- `StringCharStream` (See page 193) that takes the characters from a string
- `RandomFileCharStream` (See page 195) that takes the characters from a file.

s

191a ⟨`AbstractCharStream` *class definition* 191a⟩≡

```
CLASS AbstractCharStream;

    DEFERRED METHOD Position: INTEGER;
    -- POST
    -- NO_CHANGE,
    -- RESULT >= 0;

    DEFERRED METHOD SetPosition (NewPos: INTEGER);
    -- POST
    -- Position = NewPos;

    ⟨Computing the actual line and column number, definition 192b⟩
    ⟨Read deferred method 191b⟩

    DEFERRED METHOD LineCount: INTEGER;

    DEFERRED METHOD GetImage (Position,
                              Len: INTEGER): ARRAY OF CHAR;
    ⟨Advancing the input pointer, definition 193a⟩

END AbstractCharStream;
```

The post-condition below expresses the fact that reading a character from an `AbstractCharStream` is not a neutral operation: it alters the current position pointer. By doing so, it mimics the common semantics of files.

`Read` returns an integer, rather than a character, to be able to deal with the `EofChar` code (See page 184) and to be able to deal with inputs structured in words longer than 8 bits.

191b ⟨`Read` *deferred method* 191b⟩≡

```
DEFERRED METHOD Read: INTEGER;
    --POST
    -- RESULT <> YLexicalAnalyzer.EofChar IMPLIES
    -- Position = OLD Position + 1;
```

192a ⟨AbstractCharStream *class implementation* 192a⟩≡
```
CLASS AbstractCharStream;

  ⟨Advancing the input pointer, implementation 193b⟩
  ⟨Computing the actual line and column number, implementation 192c⟩

END AbstractCharStream;
```

The position managed by `AbstractCharStream` and all its derived classes is always an offset relative to the begin of the input. In practice, it is often much more comfortable to be able do express this position in terms of line and column number within something that might look like a source file where lines are separated by a new line character.

However, in order to remain consistent with the rest of the YAFL world, this line and column number are computed starting at 0 rather than 1. If one whishes to use these for common compiler-like error reporting, one must add 1 to these values.

192b ⟨*Computing the actual line and column number, definition* 192b⟩≡
```
DEFERRED METHOD ComputeLineAndColumnNr(NewLine: CHAR);
METHOD SetLineAndColumnNr (LineNr, ColumnNr: INTEGER);
METHOD LineNr: INTEGER;
METHOD ColumnNr: INTEGER;
```

192c ⟨*Computing the actual line and column number, implementation* 192c⟩≡
```
VAR
  TheLineNr, TheColumnNr: INTEGER;

  METHOD LineNr: INTEGER;
    BEGIN
    RESULT ← TheLineNr;
    END LineNr;

  METHOD ColumnNr: INTEGER;
    BEGIN
    RESULT ← TheColumnNr;
    END ColumnNr;
```

It is the derived classes's responsibility to actually compute this line and column number, and to call `SetLineAndColumnNr` so that the information can be fetched afterwards. `AbstractCharStream` is rather passive here...

192d ⟨*Computing the actual line and column number, implementation* 192c⟩+≡
```
METHOD SetLineAndColumnNr (LineNr, ColumnNr: INTEGER);
  BEGIN
  TheLineNr ← LineNr;
  TheColumnNr ← ColumnNr;
  END SetLineAndColumnNr;
```

`Advances` moves the input pointer, by relying on two deferred methods, namely `Position` (See page 191) and `SetPosition` (See page 191).

193a ⟨*Advancing the input pointer, definition* 193a⟩≡
```
METHOD Advance (Offset: INTEGER);
```

193b ⟨*Advancing the input pointer, implementation* 193b⟩≡
```
METHOD Advance (Offset: INTEGER);
  BEGIN
  SetPosition (Position + Offset);
  END Advance;
```

4.13.3 `StringCharStream`

`StringCharStream` inherits from `AbstractCharStream`, and implements a stream where characters are taken from a string which is given at creation time.

It applies well to case where the input is limited in size, and typically, does not even come from a file at all (Typically, expression evaluators).

193c ⟨`StringCharStream` *class definition* 193c⟩≡
```
CLASS StringCharStream;
  INHERITS AbstractCharStream;
  REDEFINE METHOD CREATE(Str: ARRAY OF CHAR);
END StringCharStream;
```

193d ⟨`StringCharStream` *class implementation* 193d⟩≡
```
CLASS StringCharStream;
  INHERITS AbstractCharStream;

  VAR
    TheStr: ARRAY OF CHAR;

  REDEFINE METHOD CREATE(Str: ARRAY OF CHAR);
    BEGIN
    BASE;
    TheStr ← Str;
    END CREATE;

  ⟨Position management methods, as redefined by StringCharStream 194a⟩
  ⟨Read, as redefined by StringCharStream 194b⟩
  ⟨GetImage, as redefined by StringCharStream 194c⟩
  ⟨ComputeLineAndColumnNr as redefined by StringCharStream 195a⟩
  ⟨LineCount as redefined by StringCharStream 195b⟩

END StringCharStream;
```

The current position within a `StringCharStream` is managed by a simple attribute `ThePosition`.

194a ⟨*Position management methods, as redefined by* `StringCharStream` 194a⟩≡
```
VAR
  ThePosition: INTEGER;

REDEFINE METHOD Position: INTEGER;
  BEGIN
  RESULT ← ThePosition;
  END Position;

REDEFINE METHOD SetPosition (NewPos: INTEGER);
  BEGIN
  ASSERT NewPos ≥ 0;
  IF NewPos > TheStr.SIZE THEN
    ThePosition ← TheStr.SIZE;
   ELSE
    ThePosition ← NewPos;
    END;
  END SetPosition;
```

Given the way the position is handled (see above), redefining `Read` is pretty straightforward.

194b ⟨`Read`, *as redefined by* `StringCharStream` 194b⟩≡
```
REDEFINE METHOD Read: INTEGER;
  BEGIN
  ASSERT ThePosition ≥ 0;
  IF ThePosition < TheStr.SIZE THEN
    RESULT ← SYSTEM.ORD(TheStr_{ThePosition});
    ThePosition ← ThePosition + 1;
   ELSE
    RESULT ← YLexicalAnalyzer.EofChar;
    END;
  END Read;
```

Similarly, getting a token's image is merely a question of computing the proper SLICE.

194c ⟨`GetImage`, *as redefined by* `StringCharStream` 194c⟩≡
```
REDEFINE METHOD GetImage (Position,
                          Len: INTEGER): ARRAY OF CHAR;
  BEGIN
  IF (Position < TheStr.SIZE) ∧ (Len > 0) THEN
    RESULT ← TheStr.SLICE (Position, Len);
    END;
  END GetImage;
```

To compute the line and column number at a given input position, we use here a very simple strategy, where we simply count the number of newline characters. This is acceptable, given the probably limited size of the input.

In case of larger inputs, `StringCharStream` will probably not be used, and a more efficient strategy will be used (See method *YLexicalAnalyzer.RandomFileCharStream.ComputeLineAndColumnNr* on page 203).

195a ⟨`ComputeLineAndColumnNr` *as redefined by* `StringCharStream` 195a⟩≡

```
REDEFINE METHOD ComputeLineAndColumnNr(NewLine: CHAR);
  VAR
    LineNr, ColNr: INTEGER;
  BEGIN
  FOR i ∈ 0 .. Position - 1 DO
    IF TheStr_i = NewLine THEN
      LineNr ← LineNr + 1;
      ColNr ← 0;
     ELSE
      ColNr ← ColNr + 1;
      END;
    END;
  SetLineAndColumnNr (LineNr, ColNr);
  END ComputeLineAndColumnNr;
```

Counting the total number of lines in an input gets quite predictable...

195b ⟨`LineCount` *as redefined by* `StringCharStream` 195b⟩≡

```
REDEFINE METHOD LineCount: INTEGER;
  BEGIN
  RESULT ← 1;
  FOR Ch ∈ TheStr | Ch = SYSTEM.CHR(SYSTEM.NewLine) DO
    RESULT ← RESULT + 1;
    END;
  END LineCount;
```

4.13.4 `RandomFileCharStream`

`RandomFileCharStream` implements a stream where characters are read from a file. It is not optimized for memory usage, though. The entire file is read in memory, to support backtracking, that is, the ability to rollback what has been read before, and read it over and over again.

However, in order to support an hypothetic future version where only parts of the file are kept in memory at any given moment, the file is read in *fixed size pages*, rather than as a continuous buffer. Such a memory-optimized system would typically limit the number of simultaneous pages in memory.

196 ⟨RandomFileCharStream *class definition* 196⟩≡

```
CLASS RandomFileCharStream;
  INHERITS AbstractCharStream;

  REDEFINE METHOD CREATE (Input: InputStream);
  ⟨Fetching a line from the input, definition 205d⟩
END RandomFileCharStream;
```

197a ⟨RandomFileCharStream *class implementation* 197a⟩≡
```
CLASS RandomFileCharStream;
  INHERITS AbstractCharStream;

  ⟨RandomFileCharStream constants and variables 197b⟩
  ⟨CREATE as redefined by RandomFileCharStream 198b⟩

  ⟨Position management for RandomFileCharStream 201⟩

  REDEFINE METHOD Read: INTEGER;
-- POST
-- OLD Position >= Size IMPLIES RESULT = YLexicalAnalyzer.EofChar,
-- OLD Position < Size IMPLIES Position = OLD Position + 1;
    BEGIN
    IF ThePosition ≥ Size THEN
      RESULT ← YLexicalAnalyzer.EofChar;
     ELSE
      RESULT ← SYSTEM.ORD (ThePage[ThePosition / PageSize]
                               [ThePosition MOD PageSize]);
      ThePosition ← ThePosition + 1;
      END;
    END Read;

  ⟨LineCount, as redefined by RandomFileCharStream 200c⟩
  ⟨GetImage as redefined by RandomFileCharStream 202a⟩
  ⟨ComputeLineAndColumnNr as redefined by RandomFileCharStream 203b⟩
  ⟨Fetching a line from the input, implementation 206a⟩

  METHOD Size: INTEGER;
    BEGIN
    RESULT ← TheSize;
    END Size;


END RandomFileCharStream;
```

`PageSize` holds the size of the pages, expressed in number of characters.

197b ⟨RandomFileCharStream *constants and variables* 197b⟩≡
```
CONST
  PageSize = 4096;
```

`MaxPageCount` holds the maximum number of allocated pages, yielding a maximum file size of $4096 \times 4096 = 16\ MB$, which is an acceptable maximum for the total input file size a lexer must recognize.

197c ⟨RandomFileCharStream *constants and variables* 197b⟩+≡
```
  MaxPageCount = 4096;
```

ThePage actually holds the pages, TheLinePosition is an array of integer that indicate where each line begins relative to the beginning of the file, to improve the performance of ComputeLineAndColumnNr (See page 203). TheLineCount holds the number of lines in the input, while TheSize holds the total size of the input in terms of number of characters.

198a ⟨RandomFileCharStream *constants and variables* 197b⟩+≡

```
VAR
  ThePage: ARRAY OF ARRAY OF CHAR;
  TheLinePosition: ARRAY OF INTEGER;
  TheLineCount,
  TheSize: INTEGER;
```

At creation time, the input stream denoted by Input is read entirely into subsequent, fixed size pages.

198b ⟨CREATE *as redefined by* RandomFileCharStream 198b⟩≡

```
REDEFINE METHOD CREATE(Input: InputStream);
  VAR
    CurPos,
    LineNr,
    CharsRead,
    PageCount: INTEGER;
    p: ARRAY OF CHAR;
  BEGIN
  BASE;
```

ThePage is allocated with its maximum size. It will be ajusted later to the actual number of pages used when reading the file.

198c ⟨CREATE *as redefined by* RandomFileCharStream 198b⟩+≡

```
ThePage.CREATE (MaxPageCount);
⟨First, read pages and compute line count 199a⟩
⟨Compute line positions 200a⟩
END CREATE;
```

We assume that there is always at least one line, every occurrence of the newline character adds one to the line count.

199a ⟨*First, read pages and compute line count* 199a⟩≡

```
TheLineCount ← 1;
WHILE ¬ Input.Eof DO
  ASSERT PageCount < MaxPageCount;
  ThePage_PageCount.CREATE (PageSize);
  CharsRead ← Input.ReadBuffer (ThePage_PageCount, PageSize);
  p ← ThePage_PageCount;
  ⟨Count the newline characters in page p 199c⟩
  TheSize ← TheSize + CharsRead;

  -- The following assertion ensures that unless the end of
  -- the file has been reached, when asking for PageSize
  -- characters, one gets PageSize characters indeed.
  ASSERT (¬ Input.Eof) ⇒ (CharsRead = PageSize);
  PageCount ← PageCount + 1;
  END;
```

The following chunk ensures that the input ends with a new line character. Otherwise, some line-oriented computation can fail miserably.

199b ⟨*First, read pages and compute line count* 199a⟩+≡

```
IF p_(CharsRead−1) ≠ SYSTEM.CHR(SYSTEM.NewLine) THEN
  IF CharsRead  < PageSize THEN
    p_CharsRead ← SYSTEM.CHR(SYSTEM.NewLine);
    TheSize ← TheSize + 1;
   ELSE
    ThePage_PageCount.CREATE (PageSize);
    ThePage_(PageCount,0) ← SYSTEM.CHR(SYSTEM.NewLine);
    PageCount ← PageCount + 1;
    TheSize ← TheSize + 1;
    END;
  TheLineCount ← TheLineCount + 1;
  END;
```

199c ⟨*Count the newline characters in page* p 199c⟩≡

```
FOR j ∈ 0 .. CharsRead - 1 |
      p_j = SYSTEM.CHR(SYSTEM.NewLine) DO
  TheLineCount ← TheLineCount + 1;
  END;
```

We now reallocate `ThePage` to the number of pages required to read the file in memory.

199d ⟨*First, read pages and compute line count* 199a⟩+≡

```
ThePage ← ThePage.SLICE (0, PageCount);
```

We know how many lines there are in the input, we allocate `TheLinePosition` accordingly...

200a ⟨*Compute line positions* 200a⟩≡

```
TheLinePosition.CREATE (TheLineCount + 1);
```

and compute the line positions by going through the entire input, and identifying the newline characters to decide where lines start and end.

200b ⟨*Compute line positions* 200a⟩+≡

```
LineNr ← 1;
TheLinePosition_0 ← 0;
FOR q ∈ ThePage DO
  FOR k ∈ 0 .. PageSize - 1 DO
    IF q_k = SYSTEM.CHR(SYSTEM.NewLine) THEN
      IF LineNr < TheLinePosition.SIZE THEN
        TheLinePosition_LineNr ← CurPos + 1;
        LineNr ← LineNr + 1;
        END;
      END;
    CurPos ← CurPos + 1;
    END;
  END;
```

200c ⟨`LineCount`, *as redefined by* `RandomFileCharStream` 200c⟩≡

```
REDEFINE METHOD LineCount: INTEGER;
  BEGIN
  RESULT ← TheLineCount;
  END LineCount;
```

Position management is managed by a `ThePosition` integer attribute. `Position` and `SetPosition` are then redefined in a rather straightforward fashion.

201 ⟨*Position management for* `RandomFileCharStream` 201⟩≡

```
VAR
  ThePosition: INTEGER;

REDEFINE METHOD Position: INTEGER;
  POST
    RESULT ≤ Size;
  BEGIN
  RESULT ← ThePosition;
  END Position;

REDEFINE METHOD SetPosition (NewPos: INTEGER);
  BEGIN
  IF NewPos > TheSize THEN
    ThePosition ← TheSize;
   ELSE
    ThePosition ← NewPos;
    END;
  END SetPosition;
```

`GetImage` redefines a method provided by `AbstractCharStream` (See method *YLexicalAnalyzer.AbstractCharStream.GetImage* on page 191) and handles the case where an image to be extracted spans over more than a single page.

Even if it is perhaps not very useful in the normal use of this class, `GetImage` even considers the ability to extract an image that does not only go over a boundary between adjacent pages, but even, includes entire pages.

202a ⟨`GetImage` *as redefined by* `RandomFileCharStream` 202a⟩≡

```
REDEFINE METHOD GetImage (Position,
                   Len: INTEGER): ARRAY OF CHAR;
  VAR
    StartPage, StartOffset, ToCut, ChunkSize: INTEGER;
  BEGIN
  IF (Position < TheSize) ∧ (Len > 0) THEN
    ASSERT Position ≥ 0;
    ASSERT Len > 0;
    ASSERT Position + Len ≤ TheSize;
    StartPage ← Position / PageSize;
    StartOffset ← Position MOD PageSize;
    IF StartOffset + Len ≤ PageSize THEN
      ⟨Extract RESULT from the current page only 202b⟩
     ELSE
      ToCut ← Len;
      ⟨Extract the remaining of the current page 202c⟩
      WHILE ToCut > 0 DO
        StartPage ← StartPage + 1;
        ⟨Extract from page number StartPage, and update ToCut 203a⟩
        END;
      END;
    END;
  END GetImage;
```

If the requested image is within a single page, extracting it is no more than a simple `SLICE`.

202b ⟨*Extract* `RESULT` *from the current page only* 202b⟩≡

```
RESULT ← ThePage_{StartPage}.SLICE (StartOffset, Len);
```

Before scanning pages until the required size has been extracted, first extract the remaining of the current page.

202c ⟨*Extract the remaining of the current page* 202c⟩≡

```
ChunkSize ← PageSize - StartOffset;
ASSERT ChunkSize ≥ 0;
RESULT ← ThePage_{StartPage}.SLICE (StartOffset, ChunkSize);
ToCut ← ToCut - ChunkSize;
ASSERT ToCut > 0;
```

First, one must check whether the next page will fit entirely. If it does, it is concatenated onto `RESULT`, and `ToCut` is reduced by its size. Otherwise, the proper slice is extracted, and `ToCut` is set to zero to end the loop.

203a ⟨*Extract from page number* `StartPage`*, and update* `ToCut` 203a⟩≡
```
IF ToCut ≥ PageSize THEN
  RESULT ← RESULT + ThePage_StartPage;
  ToCut ← ToCut - PageSize;
 ELSE
  RESULT ← RESULT + ThePage_StartPage.SLICE (0, ToCut);
  ToCut ← 0;
  END;
```

The first version of `ComputeLineAndColumnNr` (that redefined (See page 192)) was implemented in a brute force way, by simply scanning the characters from the beginning of the file and counting the `NewLine` characters on every invocation. When used repeatedly on large inputs, its performance, or rather, its lack thereof, dictated the need for a more efficient replacement scheme.

The system is now based on a form of successive approximations, but far more efficient than a binary search: when reading the input file, `TheLinePosition` gives the start position for each line. This array implements a mapping function f that must be reversed to meet our current need, namely, given a position in the input, determine its line and column number.

To achieve this purpose, we use the fact that the f function is monotonous ($a > b \Leftrightarrow f(a) > f(b)$) and we assume that the distribution of input line lengths will be well grouped around its mean $\bar{l}$. Hence, for a large n, we have $n \times \bar{l} \approx f(n)$.

Linearity is an acceptable hypothesis. There is no reason to believe that lines will be longer of shorter on average depending on their position in the source file.

203b ⟨`ComputeLineAndColumnNr` *as redefined by* `RandomFileCharStream` 203b⟩≡
```
REDEFINE METHOD ComputeLineAndColumnNr(NewLine: CHAR);
  VAR
    Left, Right, Guess, CharWidth, CharOffset: INTEGER;
    Done: BOOLEAN;
  BEGIN
  ASSERT Position ≥ 0;
  ASSERT Position ≤ Size;
  ⟨First, take care of trivial cases so that they don't bother us 204a⟩
  IF ¬ Done THEN
    Left ← 0;
    Right ← TheLineCount - 1;
    WHILE ¬ Done DO
      ⟨Compute the approximation based on Left and Right 204b⟩
      END;
    END;
  END ComputeLineAndColumnNr;
```

This kind of code is lengthy to debug if one wants to cover all cases properly (just as [48] refers about a binary search that fails because of a $\leq$ instead of a $<$). Just to avoid spending too much time checking for limit cases, we consider them ad hoc in the following chunk.

204a ⟨*First, take care of trivial cases so that they don't bother us* 204a⟩≡
```
  ASSERT (Size > 0) ⇒ (TheLineCount > 0);
  ASSERT (TheLineCount > 2) ⇒
         (TheLinePosition_{TheLineCount−1} > 0);
  IF (TheLineCount ≤ 1) ∨ (Position < TheLinePosition_1) THEN
    SetLineAndColumnNr (0, Position);
    Done ← TRUE;
   ELSIF (Position ≥ TheLinePosition[TheLineCount − 1]) THEN
    SetLineAndColumnNr (TheLineCount − 1, Position −
                              TheLinePosition_{TheLineCount−1});
    Done ← TRUE;
    END;
```

204b ⟨*Compute the approximation based on* `Left` *and* `Right` 204b⟩≡
```
  IF Position ≥ TheLinePosition_{Right} THEN
    Guess ← Right;
    Left ← Right;
   ELSE
    ASSERT Right ≥ Left;
    ASSERT Position ≥ TheLinePosition_{Left};
    ASSERT Position ≤ TheLinePosition_{Right};
```

`CharWidth` represents the total width from line number `Left` to line number `Right` expressed as a number of characters; while `CharOffset` represents the distance between `Position` and line number `Left` expressed in characters as well.

204c ⟨*Compute the approximation based on* `Left` *and* `Right` 204b⟩+≡
```
    CharWidth ← TheLinePosition_{Right} − TheLinePosition_{Left};
    CharOffset ← Position − TheLinePosition_{Left};

    ASSERT CharOffset ≥ 0;
    ASSERT CharWidth ≥ CharOffset;
```

Let's now compute `Guess` that uses the monotonous property of the f function defined above to approximate $f^{-1}(Position)$.

Given a, b, $f(a)$ and $f(b)$, $x \geq f(a)$ and $x \leq f(b)$, we have: $f^{-1}(x) \approx a + \frac{(b-a)\times x}{f(b)-f(a)}$.

204d ⟨*Compute the approximation based on* `Left` *and* `Right` 204b⟩+≡
```
  Guess ← Left + SYSTEM.TRUNC(
      (SYSTEM.FLOAT(Right − Left) * SYSTEM.FLOAT(CharOffset)) /
            SYSTEM.FLOAT(CharWidth));
```

Force `Guess` to lie between the acceptable bounds, if a problem arises from rounding errors.

205a ⟨*Compute the approximation based on* `Left` *and* `Right` 204b⟩+≡

```
IF Guess > Right THEN
  Guess ← Right;
  END;
IF Guess < Left THEN
  Guess ← Left;
  END;
END;
```

If `Guess` is the proper result, we stop searching. Otherwise, we retry on a limited scope, just as a binary search checks whether the value we are looking for is under or above the guessed value.

205b ⟨*Compute the approximation based on* `Left` *and* `Right` 204b⟩+≡

```
IF (TheLinePosition Guess ≤ Position) ∧
    (TheLinePosition Guess+1 > Position) THEN
  SetLineAndColumnNr (Guess, Position - TheLinePositionGuess);
  Done ← TRUE;
 ELSIF TheLinePositionGuess > Position THEN
  Right ← Guess - 1;
 ELSE
  Left ← Guess + 1;
  END;
```

Extreme cases must be taken care of...

205c ⟨*Compute the approximation based on* `Left` *and* `Right` 204b⟩+≡

```
IF Left = Right THEN
  ASSERT Position ≥ TheLinePositionLeft;
  ASSERT Position < TheLinePositionLeft+1;
  SetLineAndColumnNr (Left, Position - TheLinePositionLeft);
  Done ← TRUE;
  END;
ASSERT Left ≤ Right;
```

205d ⟨*Fetching a line from the input, definition* 205d⟩≡

```
METHOD GetLine (LineNr: INTEGER): ARRAY OF CHAR;
```

`GetLine` merely computes the position and length of line numbered `LineNr`, and calls `GetImage` (See page 202) to actually fetch the corresponding line image.

206a ⟨*Fetching a line from the input, implementation* 206a⟩≡

```
METHOD GetLine (LineNr: INTEGER): ARRAY OF CHAR;
  VAR
    Pos, Len: INTEGER;
  BEGIN
  IF (LineNr ≥ 0) ∧ (LineNr < TheLineCount) THEN
    Pos ← TheLinePosition_LineNr;
    IF LineNr + 1 < TheLineCount THEN
      Len ← TheLinePosition_LineNr+1 - Pos - 1;
     ELSE
      Len ← TheLinePosition_LineNr+1 - TheSize;
      END;
    RESULT ← GetImage (Pos, Len);
    END;
  END GetLine;
```

`Comment` describes a comment, that is, a piece of the input which is recognized by the lexer, but which is not returned to the calling parser. Besides, some pieces of the input are ignored as well – white space, line breaks – and usually, not considered valuable for the user to keep track of.

`Comment` does not keep an image of the comment: it merely keeps track of the position and length of the comment, in such a way that the actual image can then be extracted from the lexical analyzer which originally produced the `Comment`.

It inherits from `Comparable` (as provided by the YAFL standard library) so that lists of `Comments` can be sorted conveniently, provided that `Comment` redefines the `IsGreater` method (See page 209).

206b ⟨`Comment` *class definition* 206b⟩≡

```
CLASS Comment;
  INHERITS Comparable;

  REDEFINE METHOD CREATE (Code, Position, Length: INTEGER);

  ⟨Trivial consultation methods, definition 207b⟩
  METHOD IncLineNr (Inc: INTEGER);
  METHOD IncColNr (Inc: INTEGER);
  METHOD SetBeginPosition (LineNr, ColNr: INTEGER);
  METHOD SetEndPosition (LineNr, ColNr: INTEGER);

END Comment;
```

207a ⟨Comment *class implementation* 207a⟩≡
```
CLASS Comment;
  INHERITS Comparable;

  VAR
    TheLineNr, TheColumnNr,
    TheEndLineNr, TheEndColumnNr,
    TheCode, ThePosition, TheLength: INTEGER;

  REDEFINE METHOD CREATE (Code, Position, Length: INTEGER);
    BEGIN
    BASE;
    TheCode ← Code;
    ThePosition ← Position;
    TheLength ← Length;
    END CREATE;

  ⟨Using positions in terms of line and column numbers 209a⟩
  ⟨Trivial consultation methods, implementation 208⟩
  ⟨Comparing Comments 209c⟩
  ⟨Turning positions into line and column numbers, implementation 210⟩

END Comment;
```

207b ⟨*Trivial consultation methods, definition* 207b⟩≡
```
METHOD Code: INTEGER;
METHOD Position: INTEGER;
METHOD Length: INTEGER;
METHOD LineNr: INTEGER;
METHOD ColumnNr: INTEGER;
METHOD EndLineNr: INTEGER;
METHOD EndColumnNr: INTEGER;
```

208 ⟨*Trivial consultation methods, implementation* 208⟩≡

```
METHOD Code: INTEGER;
  BEGIN
  RESULT ← TheCode;
  END Code;

METHOD Position: INTEGER;
  BEGIN
  RESULT ← ThePosition;
  END Position;

METHOD Length: INTEGER;
  BEGIN
  RESULT ← TheLength;
  END Length;

METHOD LineNr: INTEGER;
  BEGIN
  RESULT ← TheLineNr;
  END LineNr;

METHOD ColumnNr: INTEGER;
  BEGIN
  RESULT ← TheColumnNr;
  END ColumnNr;

METHOD EndLineNr: INTEGER;
  BEGIN
  RESULT ← TheEndLineNr;
  END EndLineNr;

METHOD EndColumnNr: INTEGER;
  BEGIN
  RESULT ← TheEndColumnNr;
  END EndColumnNr;
```

209a ⟨*Using positions in terms of line and column numbers* 209a⟩≡
```
METHOD SetBeginPosition (LineNr, ColNr: INTEGER);
  BEGIN
  TheLineNr ← LineNr;
  TheColumnNr ← ColNr;
  END SetBeginPosition;

METHOD SetEndPosition (LineNr, ColNr: INTEGER);
  BEGIN
  TheEndLineNr ← LineNr;
  TheEndColumnNr ← ColNr;
  END SetEndPosition;
```

Some lexers are used after a preprocessing pass which adds or removes leading lines and/or columns in the input file. These two methods can be used to adjust the line and column number of comments, thereby reverting the changes in position introduced by preprocessing.

209b ⟨*Using positions in terms of line and column numbers* 209a⟩+≡
```
METHOD IncLineNr (Inc: INTEGER);
  BEGIN
  TheLineNr ← TheLineNr + Inc;
  TheEndLineNr ← TheEndLineNr + Inc;
  END IncLineNr;

METHOD IncColNr (Inc: INTEGER);
  BEGIN
  TheColumnNr ← TheColumnNr + Inc;
  TheEndColumnNr ← TheEndColumnNr + Inc;
  END IncColNr;
```

`IsGreater`, as redefined here, constraints its argument `Other` to a `Comment` using a `WHAT` statement and aborts brutally if `Other` is of any other type, since it does not make sense to compare `Comments` with any other thing than other `Comments`.

209c ⟨*Comparing* `Comment`*s* 209c⟩≡
```
REDEFINE METHOD IsGreater (Other: Comparable): BOOLEAN;
  BEGIN
  WHAT Other OF
    IN Comment:
      RESULT ← (ThePosition > TAG.Position) ∨
                 ((ThePosition = TAG.Position) ∧
                  (TheLength > TAG.Length));
      END;
    END;
  END IsGreater;
```

Converting the positions in terms of character offset onto positions in terms of line and column number relies on `ComputeLineAndColumnNr` (See page 192) as provided by the `AbstractCharStream` class.

210 ⟨*Turning positions into line and column numbers, implementation* 210⟩≡

```
METHOD ComputeLineAndColNr (CharStr: AbstractCharStream);
  BEGIN
  CharStr.SetPosition (Position);
  CharStr.ComputeLineAndColumnNr(SYSTEM.CHR (SYSTEM.NewLine));
  THIS.TheLineNr ← CharStr.LineNr;
  THIS.TheColumnNr ← CharStr.ColumnNr;
  CharStr.SetPosition (Position + THIS.Length - 1);
  CharStr.ComputeLineAndColumnNr(SYSTEM.CHR (SYSTEM.NewLine));
  THIS.TheEndLineNr ← CharStr.LineNr;
  THIS.TheEndColumnNr ← CharStr.ColumnNr;
  END ComputeLineAndColNr;
```

Table 4.3: Codes to be replaced in the lexical analyzer skeleton

$B	Backtrack level symbol. If the backtracking level is dynamic, this is expanded into a `TheBackTrackLevel` attribute reference. Otherwise, it is expanded into the `BACKTRACKLEVEL` constant. Such a dual replacement scheme allows the YAFL compiler to optimize and remove dead code when referring to a constant `BACKTRACKLEVEL`.
$D	Date of generation
$M	Target module
$I	Initialization of `ResultsArray`. When dealing with a lexer for which backtracking is enabled, this code is replaced by the code that initializes `ResultsArray`. `ResultsArray` is an array indexed by state number, that indicates which are the valid codes recognized so far if the automaton is in a given state.
$K	Constant declarations for the tokens, meant to be pasted in the definition module attached to the generated lexer.
$T	Token image. This code is replaced by a set of YAFL case tags. One of such case tag is generated for each of the tokens in the lexical specification, so that a textual representation for each token can be used
$A	Automaton. This code is replaced by nested YAFL case statements, where, depending on the current state and on the character at the current input position, the current state is altered.
$S	Ancillary methods. Some C compilers do not accept very large nested `case` statements gracefully. When dealing with large lexers, if the ad hoc command-line option has been used (See table 3.1 on page 38), this code is replaced a number of methods, one for every state of the DFA.
$V	Version

4.14 The lexical analyzer's skeleton

While described using `noweb` [34], this module is no valid YAFL module. It describes the *skeletons* that are used to generate the working lexical analyzers. There are two such skeletons: one for the definition module to generate, one for the implementation module to generate.

In these modules, each code prefixed by a dollar sign "$" is replaced by a fragment of YAFL code generated by `lexyt` based on the lexical specification.

The codes that can be used with these skeletons are listed in table 4.3.

4.15 Lexical analyzer skeleton

Of course, `$M` is meant to be replaced by the name of the generated module (See table 4.3 in page 211). It is not the true module name !

⟨*Definition* ⟩≡

```
DEFINITION MODULE $M;

FROM YLexicalAnalyzer IMPORT YLexicalAnalyzer;

    ⟨Lexical analyzer definition 212⟩

END $M;
```

The analyser's definition is quite limited. It merely redefined methods inherited from its base class `YLexicalAnalyzer` (See page 183) rather than defining its own, so that automatically generated lexical analyzers can be used in a polymorphic way.

The `$K` code will be replaced by the constant declarations for the tokens of the lexical specification.

212 ⟨*Lexical analyzer definition* 212⟩≡

```
CLASS $M;
  INHERITS YLexicalAnalyzer;

  REDEFINE METHOD CREATE;

  CONST
    Date = ‘‘$D”;
    LexYtVersion = ‘‘$V”;
    $K
END $M;
```

⟨*Implementation* ⟩≡
```
IMPLEMENTATION MODULE $M;

IMPORT SYSTEM;
FROM YLexicalAnalyzer IMPORT AbstractCharStream;

--------------------------
    CLASS $M;
      INHERITS YLexicalAnalyzer;

    ⟨Lexical analyzer's attributes and constants 216a⟩

    ⟨Lexical analyzer's CREATE method 218c⟩

    ⟨CurrentToken method 217d⟩
    ⟨BackTrack method 224d⟩

    $S

    ⟨ReadCurrentToken redefinition 219⟩
    ⟨Advance method 223b⟩
    ⟨ResetMaximumLength method 224c⟩
    ⟨Image method 228a⟩
    ⟨Redefining YLexicalAnalyzer simple methods 228b⟩
    ⟨Restoring contexts 229b⟩

    ⟨Redefining AttachCharStream 223a⟩

    ⟨TokenImage redefinition 222b⟩

    END $M;

END $M;
```

At first sight, it is tempting to believe that backtracking will be a costly feature. In practice, it can be implemented in a very efficient way, by guaranteeing two essential properties:

- The actual time required by the automaton itself is not altered by the fact that backtracking is used or not
- Backtracking does not require rerunning the automaton.

In other words, the automaton builds a data structure that is visited when backtracking.

Let's show how things happen on an example. Let's consider the following lexical specification.

```
If                if
IfDef             ifdef
Ident             [a-zA-Z][a-zA-Z0-9_]+
$CLASS Any        If-Ident
```

Let's consider how the automaton behaves when given "ifdefined" as input, followed by white space or a punctation character. We'll start with the most general case, namely, when dealing with lexers with backtracking level 3.

When receiving the first character "i", the lexer goes into a state where it knows that the current input *so far, that is, as a string of one character*, can be an `Ident` or an `AnyToken`. This does not mean it will stop at this first character. It only means that the string made of single "i" character can be understood as an `Ident` or an `AnyToken`, if at some stage, we must find out what this one character string can be.

The lexer stores an array made of two integer components which values are `Ident` and `Any` in the first entry of `ArrayArrayTokens`(See page 217). It uses the first entry because it refers to the possible interpretations of the token of length 1 on the current position of the input stream.

The array (in this case, made of `Ident` and `Any`) can be statically attached to each state of the automaton. Hence, instead of being reallocated and recomputed on each iteration, it is precomputed and stored in an array `ResultsArray` (See page 217), so that the statement[6]

```
ArrayArrayTokens[CurrentLength] :=
       ResultsArray[CurrentState];
```

only requires a reference assignment. All the arrays the lexer deals with when actually analysizing an input are preallocated, thereby providing far better runtime performance than if a number of brand new allocations were required for each analyzed token.

Ident Any								

[6]The actual statement in the code is slightly more complex, but this quite clearly gives the idea

After analyzing the second character, thus having recognized “if”, the input so far holds three different interpretations, namely, `If`, `Ident` and `Any`. An array with these three values is entered in `ArrayArrayTokens` at index 2 (because we have analyzed 2 characters so far).

Ident Any	If Ident Any							

and the process goes on, storing arrays of possible interpretations of the input for each length in `ArrayArrayTokens`.

After repeating the process described here for all the characters of the input “ifdefined”, `ArrayArrayTokens` can be represented as:

								Length = 9 ↓	
Ident Any	If Ident Any	Ident Any	Ident Any	IfDef Ident Any	Ident Any	Ident Any	Ident Any	Ident Any	← Incarnation = 0

The lexer cannot fill `ArrayArrayTokens` any further because “ifdefined” is followed by a white space or punctuation. The token cannot be extended beyond its 9^{th} character. The array is entirely initialized, giving the possible interpretations for each token length smaller or equal to 9. We first consider the longest possible interpretation (here, 9) in its first incarnation.

When backtracking, we simply try the next acceptable interpretation, first by checking the next incarnation:

								Length = 9 ↓	
Ident Any	If Ident Any	Ident Any	Ident Any	IfDef Ident Any	Ident Any	Ident Any	Ident Any	Ident Any	← Incarnation = 1

Then, by checking prefixes:

							Length = 8 ↓		
Ident Any	If Ident Any	Ident Any	Ident Any	IfDef Ident Any	Ident Any	Ident Any	Ident Any	Ident Any	← Incarnation = 0

until the entire array has been scanned. This process will return the same codes (here, `Ident` and `Any`) several times. In practice, in many cases, such behaviour is not desired. If the lexical analyzer runs on backtracking level 2 instead of 3, codes are burned, that is, as soon as a given code has been returned, it cannot be returned again unless the input pointer is moved.

For instance, in our example, backtracking once will *burn* `Ident`

								Length = 9 ↓	
~~Ident~~ Any	If ~~Ident~~ Any	~~Ident~~ Any	~~Ident~~ Any	IfDef ~~Ident~~ Any	~~Ident~~ Any	~~Ident~~ Any	~~Ident~~ Any	~~Ident~~ Any	← Incarnation = 1

and backtracking one more time will burn `Any` as well

				Length = 5 ↓					
~~Ident~~ ~~Any~~	If ~~Ident~~ ~~Any~~	~~Ident~~ ~~Any~~	~~Ident~~ ~~Any~~	IfDef ~~Ident~~ ~~Any~~	~~Ident~~ ~~Any~~	~~Ident~~ ~~Any~~	~~Ident~~ ~~Any~~	~~Ident~~ ~~Any~~	← Incarnation = 0

in such a way that backtracking again will return `IfDef` rather than `Ident` and `Any` over and over again.

Burned codes are implemented by the `TheBurned` array, rather than actually patching the `ArrayArrayTokens` array as suggested above, because this array's components are shared, and must be immutable.

When dealing with backtracking level 1 where prefixes are not considered, only the rightmost entry in `ArrayArrayTokens` is maintained. Backtracking will then return the various codes that can be returned on this maximal length.

When dealing with backtracking level 0, where a single code can be returned, only the uppermost and rightmost entry of `ArrayArrayTokens` is maintained in `AtomicToken`. In this case, `lexyt` behaves similarly to `lex` [40].

`TheImage` holds the image of the current token after that the `Image` method has been called, so that subsequent calls do not recompute this image. This attribute is reset everytime a new token is read.

216a ⟨*Lexical analyzer's attributes and constants* 216a⟩≡

```
VAR
  TheImage: ARRAY OF CHAR;
```

`TheMaximumLength` denotes the length of the longest incarnation of the current token. If the backtracking level is 0 or 1, it is always equal to `TheCurrentLength` since no prefixes are considered.

`TheCurrentLength` denotes the current length of the token. On analyzers with backtracking levels of 2 or 3, by backtracking, valid prefixes will be considered.

216b ⟨*Lexical analyzer's attributes and constants* 216a⟩+≡

```
TheMaximumLength,
TheCurrentLength: INTEGER;
```

`TheCurrentIncarnation` denotes the current incarnation of the current token *within the current length*: when considering prefixes of the current token, `TheCurrentIncarnation` is reset.

Basically, backtracking means trying another incarnation.

217a ⟨*Lexical analyzer's attributes and constants* 216a⟩+≡
```
TheCurrentIncarnation: INTEGER;
```

`AtomicToken` denotes the current token value as recognized by the analyzer. This attribute is used by all lexical analyzers, regardless of their backtracking levels.

`ArrayTokens` is significant for all lexical analyzers with backtracking level greater than 0. It denotes all the token values that can represent the current token at the current length. It is meant to be indexed by `TheCurrentIncarnation`.

`ArrayArrayTokens` is significant for all lexical analyzers of backtracking level 2 and 3. It represents all the token values that can represent the current token and its prefixes, indexed by `TheCurrentLength`.

217b ⟨*Lexical analyzer's attributes and constants* 216a⟩+≡
```
AtomicToken: INTEGER;
ArrayTokens: ARRAY OF INTEGER;
ArrayArrayTokens: ARRAY OF ARRAY OF INTEGER;
```

`ErrorState` is used as a phony state number to indicate that an error has occurred. The actual value is exotic enough not to clash with normal state numbers, numbered starting at 0.

217c ⟨*Lexical analyzer's attributes and constants* 216a⟩+≡
```
CONST
  ErrorState = -9876541;
```

The `CurrentToken` method is used to consult the current token value. The alternate representations (see above) cannot be consulted directly. To get the various possible interpretations of the current token, one must ask the lexer to backtrack.

217d ⟨`CurrentToken` *method* 217d⟩≡
```
REDEFINE METHOD CurrentToken: INTEGER;
  BEGIN
  RESULT ← AtomicToken;
  END CurrentToken;
```

For lexers that must be able to backtrack, `ResultsArray` holds the precomputed arrays of possible interpretations of the input, indexed by DFA state. In other words, when in DFA state n, the possible codes of the input so far can be found in $\mathit{ResultsArray}[n]$.

217e ⟨*Lexical analyzer's attributes and constants* 216a⟩+≡
```
VAR
  ResultsArray: ONCE ARRAY OF ARRAY OF INTEGER;
```

For level 2 lexers, `TheBurned` keeps track of which lexemes have already been returned at the current position, in order to prevent the lexer from returning one of these codes for prefixes of the current lexeme.

217f ⟨*Lexical analyzer's attributes and constants* 216a⟩+≡
```
TheBurned: ARRAY OF BOOLEAN;
```

If the lexer's backtracking level is dynamic, `TheBackTrackLevel` denotes the current backtracking level.

218a ⟨*Lexical analyzer's attributes and constants* 216a⟩+≡

```
TheBackTrackLevel: INTEGER;
```

`TheEofReached` indicates whether the end of the input has been reached.

218b ⟨*Lexical analyzer's attributes and constants* 216a⟩+≡

```
TheEofReached: BOOLEAN;
```

`MaxLen` is the maximum length of a token for lexers with backtracking levels equal to 2 or greater. This size is used to dimension the array that will indicate the possible interpretations of all the current token's prefixes. `MaxLen` is initialized with a size of 16k.

218c ⟨*Lexical analyzer's CREATE method* 218c⟩≡

```
REDEFINE METHOD CREATE;
  CONST
    MaxLen = 16*1024;
  BEGIN
  TheBackTrackLevel ← BACKTRACKLEVEL;
  BASE;
  ⟨Check whether the entire result arrays must be computed 218d⟩
  IF ($B > 1) ∨ DYNAMICBACKTRACKLEVEL THEN
    ArrayArrayTokens.CREATE (MaxLen);
    END;
  ⟨Allocate TheBurned if required 218e⟩
  END CREATE;
```

If the backtracking level is greater than 0, or dynamic in which case one must support the possibility for this backtracking level to hold any value, including some greater than 0, `ResultsArray` must be initialized.

218d ⟨*Check whether the entire result arrays must be computed* 218d⟩≡

```
IF ($B > 0) ∨ DYNAMICBACKTRACKLEVEL THEN
  IF ResultsArray = Λ THEN
    $I
    END;
  END;
```

If the lexer is of backtracking level 2 or dynamic (that is: it might in some circumstances have to behave as a level 2 lexer), then `TheBurned` must be allocated.

Each of `TheBurned`'s entries indicates whether a given token code has already been returned or not. Hence, its size is computed as the highest token code for this lexer.

218e ⟨*Allocate* `TheBurned` *if required* 218e⟩≡

```
IF ($B = 2) ∨ DYNAMICBACKTRACKLEVEL THEN
  TheBurned.CREATE (LASTTOKEN);
  END;
```

`ReadCurrentToken` actually performs the lexical analysis by using the input of the currently attached character stream to move through the DFA.

Careful analysis of this code shows that it does not allocate any memory by itself. It reuses existing and already allocated data structures (Namely, `ArrayTokens`(See page 217), `ArrayArrayTokens`(See page 217), `ResultsArray`(See page 217), and `TheBurned`(See page 217)) to store the information describing the lexemes that have been recognized so far, and does not even allocate or even store anything for the image of the token, leaving this to another method (Namely, `Image`(See page 228)) so that this image is only allocated if required.

219 ⟨`ReadCurrentToken` *redefinition* 219⟩≡

```
VAR
  State,
  AnalyzedSize: INTEGER;

REDEFINE METHOD ReadCurrentToken;
  VAR
    GoOn: BOOLEAN;
    Ch: INTEGER;
    OldPos: INTEGER;
  BEGIN
  IF $B = 2 THEN
    ⟨Fill TheBurned with FALSE 220a⟩
    END;
  TheCurrentLength ← 0;
  TheCurrentIncarnation ← 0;
  State ← 0;
  AnalyzedSize ← 0;
  ⟨Save initial position in OldPos 220b⟩
  GoOn ← TRUE;
  AtomicToken ← ERROR;
  ⟨Lexer's loop 220d⟩
  ⟨Restore initial position stored in OldPos 220c⟩
  END ReadCurrentToken;
```

When starting the DFA, we consider that all previously burned tokens can be valid again. Hence, we reset the `TheBurned` array. Of course, this is only required if the lexical analyzer's backtracking level is 2 or dynamic. YAFL

In early profiling of the generated lexers, this tight loop used a significant amount of time. More specifically, its statement count was very high, even if it might have been far less dramatic in terms of real performance because the actual statement (namely, an assignment) is a very primitive one.

However, in order to improve the overall performance and this biased measure at the same time, the YAFL compiler has been patched so that such structures can be detected statically and replaced by an optimized string fill function call (See page 973).

220a ⟨*Fill* TheBurned *with* FALSE 220a⟩≡
```
FOR i ∈ 0 .. TheBurned.SIZE - 1 DO
  TheBurned_i ← FALSE;
  END;
```

Since ReadCurrentToken should not change the current input pointer, it first saves the current position in OldPos, and then restore it before returning.

220b ⟨*Save initial position in* OldPos 220b⟩≡
```
OldPos ← GetCharStream.Position;
```

220c ⟨*Restore initial position stored in* OldPos 220c⟩≡
```
IF GetCharStream.Position ≠ OldPos THEN
  TheEofReached ← FALSE;
  GetCharStream.SetPosition(OldPos);
  END;
```

220d ⟨*Lexer's loop* 220d⟩≡
```
WHILE GoOn DO
  ⟨Read next character 220e⟩
  ⟨Keep track of state if backtracking is enabled 221a⟩
  CASE State OF
    ErrorState:
     ⟨Stop analyzer 221b⟩
     END;

    ⟨State handling generated by lexyt 222a⟩

    END; -- Case
  AnalyzedSize ← AnalyzedSize + 1;
  END; -- While
```

220e ⟨*Read next character* 220e⟩≡
```
IF TheEofReached THEN
  State ← ErrorState;
 ELSE
  Ch ← GetCharStream.Read;
  IF Ch = EofChar THEN
    TheEofReached ← TRUE;
    END;
  END;
```

If backtracking is enabled, the semantic actions in the automaton (that basically give a value to AtomicToken) are disabled by conditional compilation if the backtracking level is set statically, or dynamically otherwise.

In such cases, this is where it all happens. Testing for a positive state in `STATE >= 0` excludes `ErrorState` (See page 217).

221a ⟨*Keep track of state if backtracking is enabled* 221a⟩≡

```
IF $B > 0 THEN
  IF (State ≥ 0) ∧ (ResultsArray_State ≠ Λ) THEN
    TheCurrentLength ← AnalyzedSize;
    IF $B = 1 THEN
      ArrayTokens ← ResultsArray_State;
      AtomicToken ← ArrayTokens_0;
     ELSIF $B > 1 THEN
      ArrayTokens ← ResultsArray_State;
      ArrayArrayTokens_TheCurrentLength ← ArrayTokens;
      AtomicToken ← ArrayTokens_0;
      END;
   ELSIF $B > 1 THEN
    ArrayArrayTokens_AnalyzedSize ← Λ;
    END;
  END;
```

When there is no more valid transition, whatever has been recognized so far is retained, as describing the current token, or, in case of backtracking lexer, the set of tokens that can be seen at the current position within the input.

If the analyzed length is 0, specific action must be taken. If an error code must be returned, the current token length is set to 1, so that further call to `Advance` will skip the offending character, and retry synchronizing the lexer on the next one.

221b ⟨*Stop analyzer* 221b⟩≡

```
GoOn ← FALSE;
IF (TheCurrentLength = 0) THEN
  IF TheEofReached THEN
    AtomicToken ← EOF;
   ELSE
    TheCurrentLength ← 1;
    AtomicToken ← ERROR;
    END;
  END;
```

The code `$A` is replaced (See page 131) ... by code which, depending on the current state, and the current character, moves the automaton to the appropriate state:

```
2:
CASE Ch OF
0 TO 33,35 TO 255:
  State := 244; END;
34:
  State := 245; END;
ELSE State := ErrorState; END;
END;
3:
CASE Ch OF
0 TO 38,40 TO 255:
  State := 242; END;
39:
  State := 243; END;
ELSE State := ErrorState; END;
END;
```

222a ⟨*State handling generated by* `lexyt` 222a⟩≡

```
  $A    -- Nested cases, generated automatically
```

The `TokenImage` method redefines the deferred version in the base class `YLexicalAnalyzer` (See page 187) This version as redefined here returns a textual representation corresponding to a token code `TokenNr`. It handles the cases `ERROR` and `EOF` that are common to all lexers, and use the `$T` code to let `lexyt` paste the code for other tokens.

222b ⟨`TokenImage` *redefinition* 222b⟩≡

```
  REDEFINE METHOD TokenImage (TokenNr: INTEGER): ARRAY OF CHAR;
    BEGIN
    CASE TokenNr OF
      ERROR:
        RESULT ← ‘‘ERROR”;
        END;
      EOF:
        RESULT ← ‘‘EOF”;
        END;
  $T
     ELSE
      RESULT ← ‘‘<Unknown>”;
      END;
    END TokenImage;
```

`AttachCharStream`, well, attaches a character stream to the lexer denoted by `THIS`. This character stream will be split into lexemes by the lexer denoted by `THIS`. Several attributes must be reset, just as if the current position within that char stream had been changed.

223a ⟨*Redefining* `AttachCharStream` 223a⟩≡
```
REDEFINE METHOD AttachCharStream
                    (CharStream: AbstractCharStream);
  BEGIN
  BASE(CharStream);
  TheCurrentLength ← 0;
  TheEofReached ← FALSE;
  TheImage ← Λ;
  END AttachCharStream;
```

`Advance` skips what it considers as the current token (by skipping the corresponding number of characters), and goes on until a significant token is returned, that is, until it returns another code than `SKIP` (See page 185).

223b ⟨`Advance` *method* 223b⟩≡
```
REDEFINE METHOD Advance;
  VAR
    GoOn: BOOLEAN;
  BEGIN
  IF TheCurrentLength > 0 THEN
    GetCharStream.Advance (TheCurrentLength);
    END;
  GoOn ← TRUE;
  TheImage ← Λ;
  WHILE GoOn DO
    ReadCurrentToken;
    ⟨Consider skippable tokens 224a⟩
    END;
  ⟨Burn the current token if needed 224b⟩
  ASSERT AtomicToken ≥ 0;
  END Advance;
```

Codes larger than SKIPOFFSET typically denote white space and comments. Hence, when recognizing one of these codes, the current position is advanced, and ReadCurrentToken (See page 219) is called again until a significant – understand: not a white space – lexeme is recognized.

224a ⟨*Consider skippable tokens* 224a⟩≡
```
IF AtomicToken > SKIPOFFSET THEN
  Skip (AtomicToken);
  GetCharStream.Advance (TheCurrentLength);
 ELSE
  GoOn ← FALSE;
  IF $B > 1 THEN
    TheMaximumLength ← TheCurrentLength;
    END;
  END;
```

If the generated lexer's backtracking level is 2, the current token must be *burned*, that is, one must make sure that this code will not be returned again as long as we are considering the current position.

224b ⟨*Burn the current token if needed* 224b⟩≡
```
IF $B = 2 THEN
  IF AtomicToken < LASTTOKEN THEN
    TheBurned_AtomicToken ← TRUE;
    END;
  END;
```

Not much to say, I'm afraid. ResetMaximumLength simply resets the backtracking pointers, as if ReadCurrentToken had been invoked on the current position.

224c ⟨ResetMaximumLength *method* 224c⟩≡
```
REDEFINE METHOD ResetMaximumLength;
  BEGIN
  IF $B > 1 THEN
    TheCurrentLength ← TheMaximumLength;
    TheCurrentIncarnation ← 0;
    END;
  END ResetMaximumLength;
```

BackTrack walks through the data structure that has been built when the ReadCurrentToken method was invoked, and sets the current code of the lexer accordingly.

Different action must be taken, depending on the backtracking level of the lexer.

224d ⟨BackTrack *method* 224d⟩≡
```
REDEFINE METHOD BackTrack;
  BEGIN
  IF (AtomicToken ≠ ERROR) THEN
```

For instance, if the lexer supports no backtracking at all, calling `BackTrack` yields `ERROR` (See page 185).

225a ⟨`BackTrack` *method* 224d⟩+≡
```
IF ($B = 0) ∨ (AtomicToken = EOF) THEN
   AtomicToken ← ERROR;
  ELSE
   ⟨Check next incarnation 226b⟩
   END;
END;
```

On the other hand, when we deal with lexers that must be able to consider prefixes (backtracking level 2 or 3), a special case must be handled. If the current position supports at least one non-SKIP incarnation and a SKIP incarnation on a proper prefix, when getting back on this prefix, the lexer must `Advance` to skip this prefix, because due to the very semantic attached to the SKIP code, the calling syntactical analyzer cannot support receiving a SKIP.

In practice, this problem showed in the following example. Let's consider `~B` as denoting the beginning of a line. In some places, we want to be able to recognize an identifier as starting a line, but in many other circumstances, we would rather ignore this beginning of line information.

```
Ident         [a-zA-Z][a-zA-Z0-9_]*
BolIdent      ~B{Ident}
_Skip         ~B
```

`BolIdent` denotes such an identifier that starts a line. Let's imagine that we would like to make a transition on an `Ident`, no matter where it stands, at the beginning of the line or not. Bad luck, the identifier that is on the input at that stage happens to be at the beginning of the line. Quite logically, the lexer will return `BolIdent`. Since the parser rather expects an `Ident`, the lexer will backtrack. It must then realize that its valid prefix `~B` is *skippable*, actually skip it, and return `Ident`.

225b ⟨`BackTrack` *method* 224d⟩+≡
```
IF ($B ≥ 2) THEN
   IF (AtomicToken > SKIPOFFSET) THEN
     Skip (AtomicToken);
     TheCurrentIncarnation ← 0;
     Advance;
    ELSIF ($B=2) THEN
     ⟨Consider exclusion 226a⟩
     END;
   END;
END BackTrack;
```

Exclusion is a mechanism supported by `lexyt` (See page 35) to indicate that after having recognized a given token, by backtracking, some other code ought to be excluded, as in:

```
EndIf  END\-IF
End    END

$EXCLUDE EndIf End
```

where we indicate that after having recognized the keyword `END-IF`, we reject the possible recognition of a prefix of this keyword as `END`. This feature is implemented by storing the code in `ResultsArray` (See page 217) as if it was a possible interpretation, but *negating it*, to indicate that it should not be considered as a possible interpretation of the input by backtracking, but it should be *burned* instead.

226a ⟨*Consider exclusion* 226a⟩≡

```
IF (AtomicToken < 0) THEN
  TheBurned_{-AtomicToken} ← TRUE;
  BackTrack;
 ELSE
  ⟨Check for burned tokens 227b⟩
  END;
```

We first see whether a different incarnation is defined for the current length. If all incarnations at the current length have been considered, the lexer returns `ERROR` if the lexer's backtracking level is 1 (that is, if no prefixes should be tried) or attempts to find another possible interpretation among the token's prefixes.

226b ⟨*Check next incarnation* 226b⟩≡

```
TheCurrentIncarnation ← TheCurrentIncarnation + 1;
IF TheCurrentIncarnation < ArrayTokens.SIZE THEN
  AtomicToken ← ArrayTokens_{TheCurrentIncarnation};
 ELSIF $B = 1 THEN
  AtomicToken ← ERROR;
 ELSE
  ⟨Consider a prefix 227a⟩
  END;
```

When considering a prefix of the current token, we decrement the length until at least a valid token with this length can be recognized at the current position.

227a ⟨*Consider a prefix* 227a⟩≡

```
TheCurrentLength ← TheCurrentLength - 1;
TheCurrentIncarnation ← 0;
WHILE (TheCurrentLength > 0) ∧
      (ArrayArrayTokens_TheCurrentLength = Λ) DO
  TheCurrentLength ← TheCurrentLength - 1;
  END;
IF TheCurrentLength ≤ 0 THEN
  AtomicToken ← ERROR;
 ELSE
  ArrayTokens ← ArrayArrayTokens_TheCurrentLength;
  AtomicToken ← ArrayTokens_0;
  END;
```

If the current code is burned, `BackTrack` is called recursively to attempt to find another code which is not burned for the current position within the input stream. Otherwise, this very code is burned so that subsequent calls to backtrack do not return it without changing the current position.

A very smart YAFL (and even perhaps `C`) compiler might optimize the recursive call into a tail recursion jump, but it would probably not be worth the trouble anyway. Would this prove to be a potential problem, it would be much easier to derecursify the algorithm rather than try to build a universal even if seldom used scheme in the compiler. Besides, optimizing tail recursion in procedural languages only applies to cases so ideal one can question whether it is worth to invest the kind of effort it takes to actually implement such an optimization.

227b ⟨*Check for burned tokens* 227b⟩≡

```
IF AtomicToken ≠ ERROR THEN
  IF AtomicToken < 0 THEN
    TheBurned_-AtomicToken ← TRUE;
    BackTrack;
   ELSIF TheBurned_AtomicToken THEN
    BackTrack;
   ELSE
    TheBurned_AtomicToken ← TRUE;
    END;
  ASSERT AtomicToken ≥ 0;
  END;
```

`Image` returns a textual representation of the current token. It uses an attribute `TheImage` (See page 216) so that subsequent calls to `Image` without altering the position on the input stream nor the token length do not require new allocations.

228a ⟨`Image` *method* 228a⟩≡
```
REDEFINE METHOD Image: ARRAY OF CHAR;
  BEGIN
  IF (TheImage = Λ) ∨ (TheImage.SIZE ≠ TheCurrentLength) THEN
    TheImage ← GetCharStream.GetImage (GetCharStream.Position,
                                       TheCurrentLength);
    ASSERT (TheImage = Λ) ∨ (TheImage.SIZE = TheCurrentLength);
    END;
  RESULT ← TheImage;
  END Image;
```

228b ⟨*Redefining* `YLexicalAnalyzer` *simple methods* 228b⟩≡
```
REDEFINE METHOD CurrentLength: INTEGER;
  BEGIN
  RESULT ← TheCurrentLength;
  END CurrentLength;
```

228c ⟨*Redefining* `YLexicalAnalyzer` *simple methods* 228b⟩+≡
```
REDEFINE METHOD SetBackTrackLevel (Level: INTEGER);
  BEGIN
  TheBackTrackLevel ← Level;
  END SetBackTrackLevel;
```

228d ⟨*Redefining* `YLexicalAnalyzer` *simple methods* 228b⟩+≡
```
REDEFINE METHOD CurrentPosition: INTEGER;
  BEGIN
  RESULT ← GetCharStream.Position;
  END CurrentPosition;
```

228e ⟨*Redefining* `YLexicalAnalyzer` *simple methods* 228b⟩+≡
```
REDEFINE METHOD LastToken: INTEGER;
  BEGIN
  RESULT ← LASTTOKEN;
  END LastToken;
```

228f ⟨*Redefining* `YLexicalAnalyzer` *simple methods* 228b⟩+≡
```
REDEFINE METHOD CurrentIncarnation: INTEGER;
  BEGIN
  RESULT ← TheCurrentIncarnation;
  END CurrentIncarnation;
```

When changing the position in the input, one must not forget to invalidate the value stored in `TheImage`, which applied to the previous position.

229a ⟨*Redefining* `YLexicalAnalyzer` *simple methods* 228b⟩+≡

```
REDEFINE METHOD SetCurrentPosition (Pos: INTEGER);
  BEGIN
  IF Pos ≠ CurrentPosition THEN
    GetCharStream.SetPosition (Pos);
    TheImage ← Λ;
    END;
  END SetCurrentPosition;
```

When restoring a context, that is, forcing the lexer to consider a given length and/or a given incarnation, a brute force method cannot be used if the lexer's backtracking level is 2. If this is the case, `BackTrack` must be invoked to *burn* tokens that are skipped (understand, ignored) by forcing a length or an incarnation.

229b ⟨*Restoring contexts* 229b⟩≡

```
REDEFINE METHOD SetCurrentIncarnation (Carn: INTEGER);
  BEGIN
  IF $B = 2 THEN
    WHILE (TheCurrentIncarnation < Carn)
           ∧ (AtomicToken ≠ ERROR) DO
      BackTrack;
      END;
   ELSE
    TheCurrentIncarnation ← Carn;
    END;
  END SetCurrentIncarnation;
```

229c ⟨*Restoring contexts* 229b⟩+≡

```
REDEFINE METHOD SetCurrentLength (Len: INTEGER);
  BEGIN
  IF $B = 2 THEN
    ASSERT (TheCurrentLength ≥ Len) ∨ TheEofReached;
    WHILE (TheCurrentLength > Len) ∧ (AtomicToken ≠ ERROR) DO
      BackTrack;
      END;
   ELSE
    TheCurrentLength ← Len;
    END;
  END SetCurrentLength;
```

ReSync simply calls the methods described above to change the current position and to force a length and an incarnation.

230 ⟨*Restoring contexts* 229b⟩+≡

```
REDEFINE METHOD ReSync (Pos, Len, Carn: INTEGER);
  BEGIN
  IF (Pos ≠ CurrentPosition) ∨ (Len ≠ CurrentLength) ∨
     (Carn ≠ CurrentIncarnation) THEN
    GetCharStream.SetPosition (Pos);
    TheEofReached ← FALSE;
    TheImage ← Λ;
    ReadCurrentToken;
    IF $B=2 THEN
      IF AtomicToken < LASTTOKEN THEN
        TheBurned_AtomicToken ← TRUE;
        END;
      END;
    SetCurrentLength (Len);
    SetCurrentIncarnation (Carn);
    END;
  END ReSync;
```

4.16 Conclusions on lexical analysis

Here comes the end of what there was to say about lexical analysis, except for a partial review of the related works in the literature which is given together with syntactical analysis in 9 on page 947.

The model commonly depicted in  and most compiler handbooks for lexical analysis has been extended significantly. Lexical backtracking, together with all its implications, including lexical exclusion, different levels of backtracking, and efficient implementation strategies has been described extensively.

The implementation provided here is naive in many respects, but `lexyt` has proven to be a very useful and versatile tool as it is, and I truly don't believe that spending much time in making the algorithms smarter would have yielded any advantage worth mentioning. On the other hand, it would have made this document harder to write and much harder to read.

We have used `lexyt` to produce dozens of lexers already. Most of them are developed to interact with `DURA`-generated parsers (see the following chapters), but some have been used in isolation, as part of a tool which does not require language recognition mechanisms beyond lexical analysis.

`lexyt` scales very well: it routinely produces lexers with a DFA of over 2500 states. It has proven extremely stable over the years, even when significant new features were added. It can handle rather complex lexical definitions, such as the following fragment which describes the argument of a `COBOL PICTURE` clause.

```
INumber                  (\-|\+|@)[0-9]+
$VAR EXPONENT        (E{INumber})
$VAR RepeatClause        ((\([0-9]+\))|@)
$VAR PC                  ([\\$\*BPXS9VZbpxsvz\/\-\+])

$VAR PicClause001          (({PC}{RepeatClause})+)
$VAR PicClause002          ([\,\.]{PicClause001})
$VAR PicClause003          ({PicClause001}({PicClause002}*))
$VAR PicClause004          (\-|\+|@)({PicClause003})
$VAR PicClause005          (\-|\+)({RepeatClause})

$VAR PicClausePostfix          ({EXPONENT}|(CR)|(DB)|@)

PicClause     ({PicClause004}|{PicClause005})({PicClausePostfix})
```

`lexyt` provides the solid foundation on which we can now build an – hopefully – equally versatile syntactical analysis toolset.

Chapter 5

Syntactical analysis

5.1 Introduction

After having described lexical analysis in a very generic sense, and, more specifically, a complete implementation of a lexical analyzer generator, we will now look at the next layer, which will use the lexers generated by `lexyt` to produce versatile parsers.

Writing parsers is considered as a painful process for which tools exist and should be used. This holds for parsers built on top of a simplistic view of lexical analysis but this is even more the case when dealing with our versatile lexers which support backtracking.

The tool we will describe now to fulfill complex parser generation needs is called `DURA`[1]. Just as for `lexyt`, the complete and highly documented source code for `DURA` is included in this document[2].

Before going to the gory details, let's have a look at the objectives I was pursuing, the issues I wanted to address, when developing `DURA`.

5.1.1 Size

The first issue is size, both in terms of the size of the grammar, and the size of the input one deals with. In theory, the common compiler-compiler model can be extended at will, and can deal with inputs of arbitrary size. In practice, however, dealing with a monolithic grammar can be unmanageable as soon as the number of rules grows beyond a few hundred. Some of the grammars we want to produce parsers for are made of over 2000 rules, even if it is true that depending on the expressivity of the grammar rules, the actual number of rules and even non-terminals involved in these rules can change dramatically.

[1] The name `DURA` is not an acronym, it is just meant to fill the remaining gap in the famous sequel of Unix tools "*dura lex sed lex*".

[2] I have troubles believing you might not have noticed, since the `DURA` source code accounts for over 60% of the few pounds of literature you are currently holding in your hands, cursing me for having produced such a huge and undigested piece of work.

When dealing with grammars of such sizes, we must reach the highest possible level of abstraction. We must be able to describe complex grammar rules in the most expressive formalism possible. Of course, in theory, an equivalent result can be achieved using additional grammar rules, but when reaching a given size, the mechanical addition of such grammar rules for the sake of an inadequate parsing technology can be dreadful (See [49] and [19] for two equally flamboyant essays on the issue of the useless strain put on the development of parsers by inadequate parsing technology).

The size of input also makes some parsing technologies totally unsuitable for a given purpose. A Prolog [50] or Prolog-like parsing method based on immediate recursion and backtracking can be applied successfully to analyze inputs in the world of natural language recognition, where no input should be longer than a few dozen words. How does such a technique scale, when dealing with inputs of over 100000 tokens? `RainCode` (See page 920) is somehow the true final purpose of this work, and must have ways of dealing with real-world `COBOL` or `PL/1` programs, some of which are as long as 100000 lines.

5.1.2 Performance

Many – if not most – of the applications that require parsing technology today are not compilers in the usual compile-link-execute sense of the word. For instance, `RainCode` (See page 920) is not even meant to be used as an interactive tool[3]. It can run in batch. It does not analyze the same source code over and over again as common compilers do. It is used punctually to go through potentially large amounts of source code, in a context where the performance of syntactical analysis is only relevant in terms of orders of magnitude.

All in all, `RainCode`'s performance requirements are not as drastic as true compilers, and experience shows that the level of performance we have reached is very adequate for any realistic purpose.

Besides, using `DURA` as described here, we now generate parsers – to be used through the `RainCode` machinery – for languages to which automatic parser generation was barely applicable (`PL/1`, for instance), making fair comparison (or any form of comparison, whatsoever) impossible. For parsing-friendlier languages (Pascal [51], Modula II [52], Oberon [53] [13], etc.), the performance of the parsers produced by `DURA` is good enough to make them suitable for compiler construction purposes. It is true that we are still far from the performance of hand-written or `yacc`-generated parsers, but given the hardware commonly available today, the resulting parsers are quite usable.

It is a common change of attitude in the compiler construction world: parsing performance is not considered as important as it used to be. Compilers used to be slow, and since they spent most of their times in the early phases of compilation, it made sense to try to optimize them as much a possible. Issues such as those developed in [54], [55] and [56] to explain how a lexer could be made much fasters, while remaining true from a strictly scientific point of view, are far from being as industrially relevant as they were

[3] Stated more accurately, it was not meant to be used as an interactive tool in its early design phases, but new developments and ever increasing power of commonly available hardware make it now possible.

10 or 15 years ago.

Compilers are not the bottleneck anymore. Besides, when analyzing the runtime behaviour of GCC, [57] showed that the time spent in the front end was negligible when compared to the sophisticated algorithms used for the code optimizations required by modern hardware architectures (Pipelining, scheduling, etc.).

5.2 DURA and the `cool` model

The `cool` model, as described in [20], has been a significant source of inspiration for DURA, both from a positive and a negative point of view. I tried to retain and enhance when possible whatever seemed to be positive, and challenged whatever seemed to be potential points of failure in this first version. This is a very typical example of the *second version syndrom* as explained extensively by Brooks in [58]. The fact that both the lexical analyzer generator (`lexyt`) and the parser generator presented in this work are second versions of earlier prototypes as a factor toward quality and stability.

The points that I considered positive and retained from `cool`'s original design are:

- The overall architecture, that supported modularity and the morphism between classes and nonterminal, by extending the original YAFL compiler.
- More specifically, clearly segregating the collection of the grammar rules, and the generation of the corresponding parser, so that different parser generation strategies can be considered without having to change the front-end, that is, the extended YAFL compiler.
- Some abstraction mechanisms, namely repetitions and reverse inheritance.

On the other hand, the shortcomings in the original `cool` design and implementation that have been addressed are:

- Implementation. For reasons that are mainly due to the large number of late additions to `cool`'s original design, the existing implementation ended up being poorly structured. Furthermore, it required significant amending to adapt to the externalized code generation scheme as provided by the newest version of the YAFL compiler (See page 974). Hence, it soon appeared that it would be far more effective to reimplement our new tool from scratch rather than try to navigate through scaffoldings of historical patches.
- Grammar expressivity. Operators have been added, to allow the convenient description of sophisticated grammar rules without having to express them in terms of lower level primitive rules.
- Backtracking. `cool` only supported a very limited form of lexical backtracking, and it soon appeared that this was one of the most promising possible direction for extensions.

5.3 DURA and backtracking

A DURA-generated parser works just like any LR automaton, as described in [59], [29] and : it supports states, a *shift* and a *reduce* operation. In order to support languages beyond LR and LALR, it also supports two additional operations: an *unshift* and an *unreduce*.

A DURA-generated parser behaves like any other LR parser, except that it supports the ability of having more than one possible action (at most a shift, and zero or more reduces) in every state, given a lookahead token.

When in state where, given the lookahead token, no action can be taken, the last operation pushed onto the LR stack is undone. If this last operation was a shift, it is removed from the LR stack. If the last operation was a reduce, the slice of the LR stack that was replaced by the reduction is pushed back onto it. In both cases, the state of the lexer and parser *before* the operation (shift or reduce) is restored, so that the next available action can be considered. If no such other action exists, that is, if all possible actions in the topmost state have been considered unsuccessfully, the undoing process is repeated. If backtracking attempts to undo an operation while the LR stack is empty, it means that there was no way the input could be matched by the grammar. If the parser reaches a final state, the input has been recognized successfully.

Just as BTYACC [60] and PRECC [61] [62], and unlike Lark [63], ANTLR [64] and JavaCC [65] backtracking as supported by DURA is automatic. It is not the user's responsibility to indicate when a backtracking path must be considered.

5.3.1 Global and local ambiguities

In case of an ambiguous grammar where an input can match several valid parse trees, unlike Earley [30] or Tomita [66] (sometimes referred to as GLR in this document), DURA returns only the *first* matching tree, instead of the collection of *all* matching trees.

DURA is not aimed at natural language processing where an entire input can hold more than one syntax tree. Originally, Tomita and Earley were designed with natural language processing in mind. DURA is aimed at producing parsers for programming languages which are not ambiguous, but which can be *locally* ambiguous, in the sense that the ambiguity is lifted by the state of the input beyond the current window of visibility. Some might call it the lookahead.

5.3.2 Backtracking

In general, backtracking can have an intrinsic exponential behaviour. At first sight, it would be far preferable to go for a guaranteed polynomial time scheme (Earley is known to be cubic in its worst case) than to open the door for virtually uncontrollable behaviours due to backtracking.

In practice, it is an entire different story. Backtracking is not meant to be used constantly. Backtracking is a kind of *plan B* for the hopefully relatively few cases where plain LR parsing will not serve our purpose adequately. As explained in a nice metaphor by Jean-Eric Waroquier in [41], backtracking is just like virtual memory: it is a nice to have when needed, but one cannot depend on it continuously. Similarly, even if in an entirely different context, Philippe Seguin once compared referendum as a political tool to an airbag: it very useful in extreme cases, but it is not meant to be turned on continuously.

This optimistic view is more than an act of faith: it matches experimental results (See page 943). Similar figures led Tomita to design GLR [66] as a generalization of LR, assuming that in most cases, the efficiency of LR would deliver good results, falling back on a more generic and less efficient scheme when facing a LR ambiguity.

The identification of real world situations where a faster algorithm could be used *except in some cases* is a common path of research in parsing theory: BPS [67] [68], Tomita [66], and this work are examples of this trend. Quite logically, this is the major weakness of Earley [69] [30] and derived schemes: they can handle the widest number of cases, and therefore, provide a parsing power – with a cost in terms of performance – which is barely used in practice.

5.3.3 The relationship with Tomita

A Tomita parser (See 9.2.4.2 in page 956, or [70] for a complete description) clones LR stacks in case of conflict, and reconciles some of them later when different paths, after having recognized a fragment in different ways, converge to a common state.

This convergence can only occur in case of a globally ambiguous grammar, which is explicitely excluded from DURA's domain of application. The Tomita parser DAG turns into a tree, where only one path matches the input. The other paths lead to dead ends.

The correspondence with DURA is then clear: DURA is the backtracking version of a Tomita parser, where every path is visited in sequence rather than in parallel.

This supposed equivalence should be taken with a grain a salt: DURA *aims* at globally unambiguous languages, which this does not mean that all the grammars it will have to handle will be unambiguous (This question has shown undecidable by Floyd [71] and by Hopcroft and Ullman [29] as a consequence of Greibach's theorem). If it ever does, it will yield the first matching parse tree.

5.3.4 SLR vs LALR

One of the consequences of the integration of backtracking in DURA lies in the freedom it gave me to produce SLR [72] rather than LALR [73] [74] parsers. The latter of more powerful than the former in the sense that they can recognize a wider class of languages, but this difference of ability originally addressed forward-only parsing methods. When facing one of the rather uncommon cases where a LALR foward-only parser accepts an input which would have been refused by an equally forward-only SLR parser, DURA will just backtrack and eventually recognize the input flawlessly.

Restricting to SLR perhaps increases the rate of backtracking by a percent or two. On the other hand, a SLR parser can be generated directly from the grammar representation, while common algorithms to produce LALR parsers [73] [74] require additional and complex data structures.

All in all, it seemed a reasonable compromise to invest more time in other areas of this work than to implement a full LALR parser generator.

5.3.5 Lexical backtracking

This scheme can be extended easily to support lexical backtracking. An *unshift* basically replaces the lookahead token by the previous lookahead token, restoring the state of the parser as it was *before* shifting over the previous lookahead token.

It is a move backwards. Lexical backtracking simply means that before cancelling the current lookahead token – which implies that the parser has been unsuccessul using it in its current incarnation – the parser will try to see whether there is another possible interpretation of this lookahead token. If there is such an alternate interpretation, the parser will try to go forward again, using this alternate interpretation. If it leads to another failure, the parser will ultimately get back, and additional interpretations will be tested in turn, until one of them eventually leads to a successful interpretation of the input, or until all possible interpretations of the current token have been tested, in which case `DURA` proceeds with the unshift operation.

5.4 Describing `DURA` formally

5.4.1 Why a formal description ?

This section provides a reasonably thorough formalization of the mechanisms of `DURA`-generated parsers.

In the course of the (large) sections in literate programming forms given later in this document, various specifications in ObjectZ [75] describe the *static* properties of our classes. For instance, it describes the conditions a grammar and a push down automaton must fulfill in terms of the invariants which connect the various attributes together. Deliberately, it does not describe the operations used to build such valid data structures. This emphasis on the static components of the specification shows clearly in the way virtually everything is described in terms of constraints on attributes rather than in terms of operations which are used to build these "compliant" instances.

We will here now concentrate on the *dynamic behaviour* of `DURA`-generated parsers, that is, on sequences of events occurring to such parsers, and their effect on the data structures.

One might question the usefulness of this formalization. I believe it is essential for several reasons.

- Some of the concepts that are presented here are definitely not trivial and an informal description can lead to misunderstanding.
- The concepts that are presented here are used extensively in a formal proof at the end of this section. How formal would that proof be if its bases are not well-defined themselves ?
- They can serve as a guide through some parts of the implementation. This is not to say that the implementation mimics this specification – as a matter of fact, it deliberately does *not* in some circumstances. However, the trickiest parts of the various algorithms are better described here than in the implementation.
- This formal specification is definitely a nifty *exercice de style*. As far as I know, there has been no formal specification of the LR automaton: the entire world seems to rely on the very comprehensive but loose description given in . Since the system extends the LR automaton in several ways (by backtracking, for instance) I believe it is worth the trouble to define the basics carefully before showing how these extensions can help.

However, it would also be wrong to believe that this specification is exhaustive. I concentrated on the parts that I considered as essential, and left some parts partially unspecified or deliberately simplified. For instance, in the scope of this specification, and in the proof about parsers which never terminate, I simplified the specification to consider non-backtracking lexers only. The careful and willing reader can easily see how to extrapolate the specification as presented here to include backtracking lexers.

Integrating them in the proof is not very hard, but it multiplies the number of cases that must be taken into account, and the proof relative to the classification of looping parsers gets longer and far more involved, up to a point where it would be doubtful that anyone would grasp its intuitive foundations.

5.4.2 Describing the automaton

The $Parser$ class, as described below, describes the dynamic behaviour of a DURA-generated parser. It connects a $Lexer$ (See page 15), a $SyntacticSpecification$ (See page 296) and a LR stack together, to produce a working lexing and parsing machinery.

$Parser$

$Spec : SyntacticSpecification$ (See page 296)

$LRStack$ is defined as a sequence of $\downarrow ParserStackElement$, that is, a sequence of instances denoted by $ParserStackElement$ or one of its derived classes. Without the $\downarrow$ operator, it would just refer to $ParserStackElement$s without polymorphism.
$ActionNr$ controls backtracking, by holding the index number of the action to be taken in the current state.

$LRStack : \mathrm{seq} \downarrow ParserStackElement$ (See page 331)
$Lexer : Lexer$ (See page 15)
$ActionNr : \mathbb{N}$
$LRState : \rho PDAState$ (See page 638)

$LexWeights$ maintains a sequence of naturals, which, for each entry of the LR stack, yield the number of tokens behind it. $LexWeights$ will be used when proving properties of DURA-generated parsers. $Syms$ holds a symbol string which is build by taking terminal symbols from $ShiftStackElement$s (See page 335) and non-terminal symbols from $ReduceStackElement$s (See page 337).

$States : \mathrm{seq}\, PDAState$
$LexWeights : \mathrm{seq}\, \mathbb{N}$
$Syms : SymbolString$ (See page 476)

$\#States = \#LRStack$
$\forall i \in \mathrm{dom}\, States \bullet$
$\quad States(i) = LRStack(i).ToState$
$\#LexWeights = \#LRStack$
$\forall i \in \mathrm{dom}\, LexWeights \bullet$
$\quad LexWeights(i) = LRStack(i).LexWeight$
$\#Syms = \#LRStack$
$\forall i \in \mathrm{dom}\, Syms \bullet$
$\quad Syms(i) = LRStack(i).Sym$

do is the master operation, which, depending on the state of the parser and the current position in the input, triggers $Shift$, $Reduce$ or $BackTrack$ (which, in turn, will trigger $UnShift$ or $UnReduce$).

do

$\Delta(Lexer, LRStack, ActionNr, LRState)$

$Actions : iseq\ LRAction$ (See page 637)
$Actions = \delta LRState.Actions(Spec.Rules, Lexer.CurrentToken)$

If $ActionNr$ does not yield a valid action in this state, invoke $BackTrack$ (See below).

$ActionNr > \#Actions \Rightarrow BackTrack$

The two alternatives below reflect the fact that $LRAction$ is defined as the union of $\rho PDAState$ to represent a *shift* to a state of the automaton, and $\rho FlatRHS$ to represent a reduction.

$ActionNr \in \mathrm{dom}\ Actions \wedge Actions(ActionNr) \in \rho PDAState \Rightarrow$
$\quad Shift(Actions(ActionsNr))$
$ActionNr \in \mathrm{dom}\ Actions \wedge Actions(ActionNr) \in \rho FlatRHS \Rightarrow$
$\quad Reduce(Actions(ActionsNr))$

$BackTrack$

$\Delta(Lexer, LRStack, ActionNr)$

$Top :\downarrow ParserStackElement$
$Top = last\ LRStack$
$Top \in ShiftStackElement \Rightarrow UnShift$
$Top \in ReduceStackElement \Rightarrow UnReduce$

Shift

$\Delta(Lexer, LRStack, ActionNr)$
$TargetState? : \rho PDAState$

$NewTop : ShiftStackElement$
$LRStack' = LRStack \frown \langle NewTop \rangle$
$ActionNr' = 1$
$LRState' = TargetState?$
$NewTop.FromState = LRState$
$NewTop.ToState = TargetState?$
$NewTop.LexPosition = Lexer.Position$
$NewTop.LexLength = Lexer.Length$
$NewTop.LexCarnation = Lexer.Carnation$
$NewTop.ActionNr = ActionNr$
$Lexer.Advance$ (See page 15)

Reduce

$\Delta(LRStack, ActionNr)$
$RHS? : \rho FlatRHS$

$NewTop : ReduceStackElement$
$NewTop.FromState = LRState$
$NewTop.ActionNr = ActionNr$
$NewTop.LexPosition = Lexer.Position$
$NewTop.LexLength = Lexer.Length$
$NewTop.LexCarnation = Lexer.Carnation$

The two following clauses use variations on the *Slice* function, namely *TopSlice* (See page 978) and *BottomSlice* (See page 978).

$NewTop.TheArray = TopSlice(LRStack, \delta RHS?.Length)$
$LRStack' = BottomSlice(LRStack, \#LRStack - \delta RHS?.Length) \frown \langle NewTop \rangle$
$ActionNr' = 1$

The state the automaton moves to is controlled by the *Goto* function, as defined by the *SyntacticSpecification* class (See page 296). By construction of the LR automaton, we know that the stack is deep enough at this stage to extract an element at depth $RHS?.Length$.

$NewTop.ToState =$
$\quad Spec.Goto(LRStack(\#LRStack - \delta RHS?.Length).ToState,$
$\quad\quad RHS?.Enclosing)$
$LRState' = NewTop.ToState$

UnShift

$\Delta(Lexer, LRStack, ActionNr)$

$Top : ShiftStackElement$
$Top = last\ LRStack$
$LRStack' = front\ LRStack$
$ActionNr' = Top.ActionNr + 1$
$LRState = Top.FromState$

The following clause is an expansion – equivalent to a method invocation – of the $ReSync$ operation as defined for the $Lexer$ (See page 15) class, to restore the lexical state as it was stored when performing the matching $Shift$ operation. This expansion is similar to a method invocation, the renaming clauses ($Pos? = Top.LexPosition$, etc.) are used to set actual parameters.

$Lexer.ReSync \mid Pos? = Top.LexPosition,$
$Len? = Top.LexLength,$
$Carnation? = Top.LexCarnation$

UnReduce

$\Delta(LRStack, ActionNr)$

$Top : ReduceStackElement$
$Top = last\ LRStack$

This is where the unreduce operation actually occurs: we put the array of reduced $ParserStackElement$s back onto $LRStack$.

$LRStack' = front\ LRStack \frown Top.TheArray$
$ActionNr' = Top.ActionNr + 1$
$LRState = Top.FromState$

INIT

$LRStack = \langle\, \rangle$
$ActionNr = 1$
$Lexer.Position = 1$
$LRState = Spec.StartState$

5.5 Proving some of `DURA`'s properties

Cuius rei demonstrationem sane detexi hanc marginis exiguitas non caperet.

5.5.1 What do we wish to prove ?

It is not reasonable to consider proving `DURA` entirely. There are far too many features, exceptions, shortcuts, tricks and optimizations to make such a daunting task even remotely feasible.

On the other hand, there is at least one property that should be considered for `DURA` to be of any use. We must have a good level of confidence in its ability to complete its work, that is, to parse an input or to report an error in a finite amount of time. When dealing with forward only algorithms such as LALR [73] [74], Tomita [66] or even Earley [30], such questions do not arise. If we can ensure that every step is finite (and in case of Tomita, there are cases where it is not the case [76]), the algorithm is guaranteed to terminate, since the number of steps is bounded.

When dealing with an algorithm that backtracks, undoing what had been done before, it is worth questioning its ability to guarantee termination.

In order to do so, we shall first determine what we shall qualify as a cycle that would prevent the parser from terminating. We will then prove how the existence of these cycles imply that the grammar includes rules of a given form.

5.5.2 Cycles

A cycle is a situation where the parser never ends. The parser never ends if does not end with a valid parse tree, nor does it stop indicating an error. Hance, a cycle is characterized by a situation where a infinite number of operations occur.

The proof to classify the cases of cycles is based on the execution string, which describes the sequence of operations which occurs. The execution string is a sequence of atomic elements which are:

- $S_{f \to t}$ for a shift operation from state f onto state t, implemented by the $Shift$ operation in the specification of the $Parser$ class (See page 240).
- $S^{-1}_{t \to f}$ for an unshift operation from state t back to state f, implemented by the $UnShift$ operation.
- $R_{a,f \to t}$ for a reduce operation which moves the parser from state f to state t reducing the array a from the LR stack.
- $R^{-1}_{a,t \to f}$ for an unreduce operation which moves the parser from state t back to state f, putting the array a back onto the LR stack.

Each of these operations has an effect on the LR stack, and can leave different forms of $ParserStackElement$s on the top of it, as shown in table 5.1. In some cases, the form of $ParserStackElement$ on the LR stack is undefined, in the sense that, for instance, after executing a R^{-1}, the top element can be a $ShiftStackElement$ or a $ReduceStackElement$, but the choice is entirely deterministic. The table entries with "Any" do not indicate non-determinism, but just the fact that the nature of the operation without further information does not imply a specific class for the top LR stack element.

Table 5.1: Operations that can be performed on a LR stack

Top of the stack, before	Operation	Effect on stack depth	Top of stack, after
Any	$S_{f\to t}$	$+1$	$\mathit{ShiftStackElement}$
$\mathit{ShiftStackElement}$	$S^{-1}_{t\to f}$	-1	Any
Any	$R_{a,f\to t}$	$-\#a+1$	$\mathit{ReduceStackElement}$
$\mathit{ReduceStackElement}$	$R^{-1}_{a,t\to f}$	$+\#a-1$	Any

We'll refer to S_0 as the execution string, that is, the sequence of R, R^{-1}, S, and S^{-1} operations of a $Parser$. Quite obviously, our parser cycles if and only if there is an input for which S_0 is infinite.

It is intuitively clear that all sequences are not equally possible. For instance, a R^{-1} operation cannot follow a S operation directly, because the S operation leaves a $\mathit{ShiftStackElement}$ on top of the LR stack, while the R^{-1} operation requires a $\mathit{ReduceStackElement}$. The next paragraph further investigates the properties of possible S_0 sequences.

Definition 1 (Balanced strings)
A non-empty sequence (or string) of operations is balanced if it is of one of the following three forms:

$$S_{f\to t}S^{-1}_{t\to f} \tag{5.1}$$

$$R_{a,f\to t}R^{-1}_{a,t\to f} \tag{5.2}$$

$$S_{f\to t}\,L\,S^{-1}_{t\to f} \tag{5.3}$$

$$R_{a,f\to t}\,L\,R^{-1}_{a,t\to f} \tag{5.4}$$

where L is a sequence of one or more balanced strings.

Lemma 1
As defined in the specification of the $Parser$ class (See page 240), the LR stack and the LR state are unchanged when performing a balanced sequence of operations.

These invariants are trivial to prove when dealing with (5.1) and (5.2). By construction, the S and R operations save the LR state on the stack, and the matching S^{-1} and R^{-1} operations restore it, so the LR state is an invariant. Similarly, the effect on the LR stack of the S and R operations is undone by the matching S^{-1} and R^{-1} operations.

Preserving the LR stack implies the preservation of the lexical position, which is stored in the top element of the stack. Intuitively, it is clear that a balanced string of operation preserves the lexical position, since none of the operations in (5.2) alter the lexical position, while, in the sequence in (5.1), the lexical position is saved in the S operation and restored in the matching S^{-1} operation.

Then, the cases (5.3) and (5.4) can be proven by induction. □

Definition 2
S_1 is defined as S_0 where all balanced substrings have been removed.

The whole purpose here is to substitute S_1 for S_0, since Lemma 1 guarantees that the state of our system before and after executing any balanced string of operations is the same. Hence, using S_1 whenever we used S_0 will not affect the system's behaviour.

When dealing with a parser such as generated by `yacc`, that does not support backtracking, we have $S_0 = S_1$.

Lemma 2
S_1 does not contain any S^{-1} or R^{-1} operation.

Let's consider the first remaining occurrence of a S^{-1} or R^{-1} operation in S_1. If this first operation is an S^{-1}, then, the operation before it must have been a S since it cannot be a R: if it had been a R, the top element on the LR stack would have been a $ReduceStackElement$, which cannot precede a S^{-1}. If the operation before the S^{-1} is a S, by construction, they would form a balanced string, and should have been removed in the process of going from S_0 to S_1.

The case where the first operation in S_1 which is not a R and not a S is a R^{-1} can be handled similarly, and is left as an exercice to the reader. □

Lemma 3
When in a LR state s having $ActionNr = a$, the effect of executing a balanced string will be to be back in state s with $ActionNr = a + 1$.

The last operation of the balanced string is a R^{-1} or a S^{-1}, which restores the $ActionNr$ saved by the matching R or S operation, and increments it by 1. □

Lemma 4
The number of balanced strings that can be executed when in a state s is bounded.

This is a direct consequence of Lemma 3: whenever a balanced string is executed, the parser returns in its original state s except for $ActionNr$ which is incremented by 1. On the other hand, the precondition for the selection of $Shift$ or $Reduce$ in the Do operation for the $Parser$ class (See page 240) states that $ActionNr$ must be remain smaller or equal to the number of actions that can be performed when in state s. This excludes the occurrence of additional balanced strings in the input since a balanced string must start with a S or a R. Hence, the number of balanced strings that can be performed when in state s is bounded by this number of actions. □

Lemma 5
If S_0 is infinite, so is S_1. Besides, S_1 contains only S and R operations.

Is S_1 is finite, it ends in a state s. If S_0 is infinite, there are an infinite number of balanced sequences to be executed after having reached this state s, while Lemma 4 tells us that the number of balanced sequences that can follow a state is bound. S_1 cannot contain any S^{-1} or R^{-1} operation, as shown in Lemma 2. □

Definition 3
σ is the number of non-zero entries in $LexWeights$ (See page 240).

Formally, we have:

$$\begin{array}{|l} \sigma : Parser \mapsto \mathbb{N} \\ \hline \sigma = \#\{x \in \mathrm{dom}\, Parser.LexWeights \mid Parser.LexWeights(x) > 0\} \end{array}$$

Lemma 6
A S operation increases σ by one, while a R operation will decrease σ or leave it unchanged. A R operation that does not alter σ reduces an empty string, or reduces an array where one and only one entry has a non-zero $LexWeight$.

Since a S operation adds a $ShiftStackElement$ with weight 1 on the LR stack, the first part of the Lemma is trivial.

On the other hand, if the LR stack slice reduced by the R operation does not contain any entry with $LexWeight > 0$, thus denoting an empty string, removing the slice does not alter σ, neither does pushing the newly $ReduceStackElement$ which will have a $LexWeight$ of 0 as well.

If the array it reduces has n entries with a non-zero $LexWeight$ with $n > 0$, σ will be decreased by n (reducing the entries) and increased by one (the newly pushed $ReduceStackElement$ has $LexWeight > 0$). Hence, for this reduction to preserve σ, we must have $n = 1$. If $n > 1$, σ will be decreased. □

Lemma 7
For every S_1 sequence generated for a parser that cycles, there is an index i such that all operations after i are reductions (R) that do not alter σ.

By Lemma 5, S_1 is infinite, while from Lemma 6 the number of S operations within S_1 must remain bounded, otherwise the parser reaches the end of the input and fails or succeeds, but in any case, does not loop. After the last of these S's, there can only be R's. Since R's can only *decrease* σ or leave it unchanged (See Lemma 6 again), after a while, σ will have reached a minimum and will not evolve any longer. □

It is of course clear that the LR stack cannot shrink indefinitely while executing S_1. Hence, in a cycling parser, there are two cases to consider:

- The LR stack's size is bounded over the execution of S_1, that is, there is a limit size l such that $\#LRStack \leq l$ remains always true.
- The LR stack's size grows indefinitely while executing S_1.

These two cases will be considered separately.

5.5.3 The LR stack is not growing

Lemma 8
A cycling parser with a bound LR stack has at least one configuration of $Syms$, $Weights$, $States$ (See page 240) and lexical position on which it will come back an indefinite number of times.

This is very simple to prove: since the set of symbols, the set of possible weights – by Lemma 7, there is a stage after which no more S operation can occur – and the set of states are finite, since the set of valid stack sizes for such a cycling parser is bounded, the number of possible configurations is bounded as well. Consequently, there must be at least one configuration that occurs an infinite number of times. Lexical position is known not to evolve after some stage by Lemma 7 since R operations do not change the lexical position □.

Lemma 9
Between two occurrences of this configuration, there cannot be only reductions of RHSs of 0 *elements.*

Since reductions of RHSs of 0 elements let the LR stack grow, there is no way a sequence of such reductions only can preserve $Syms$ and $Weights$. □

Lemma 10
The sequence of operations between two occurrences of the ever recurring configuration referred to in Lemma 8 is always the same, and must contain at least a reduction of a RHS of at least 1 *element,*

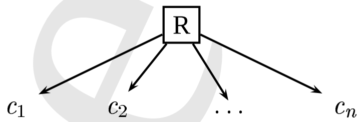
The state of the LR stack and the position in the input define the sequence of operations unambiguously. Since the LR stack (See Lemma 8) and the lexical position (See Lemma 7) are the same in all occurrences of this configuration, the sequence of subsequent operations in S_0, and thus in S_1, will be same. Besides, if there weren't at least a reduction of at least 1 element, σ would be constantly growing (Lemma 7 and Lemma 9). □

Let θ be one of the RHS's of at least one element which is reduced at least once between two of these equal configurations (See Lemma 10). θ does not alter σ (See Lemma 7).

In order to finish the proof in the case of a cycling parser with a bounded LR stack, we'll represent the behaviour of our parser executing the operations in S_1 graphically, depicting operations – shift and reduces – as trees, and the LR stack as a bush, that is, a collection of trees.

A shift operation is a childless node: $\boxed{\text{S}}$ with $S.LexWeight = 1$.

A reduce operation is a node that has n children nodes, with n being the length of the attached RHS:



The c_i subnodes of a $\boxed{\text{R}}$ node are childless $\boxed{\text{S}}$ nodes or $\boxed{\text{R}}$ nodes with or without children.

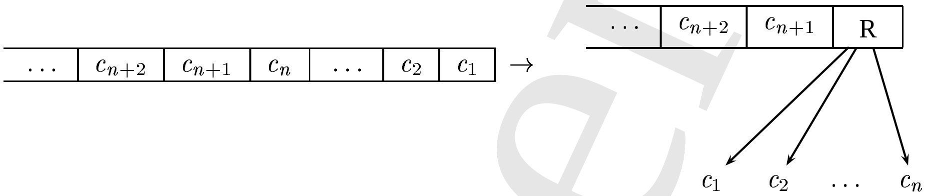
The $LexWeight$ attached to this reduce operation is of course computed as:

$$R.LexWeight = \sum_{i \in 1..n} c_i.LexWeight$$

Given these basic blocks, our parser can be described as follows: the LR stack is just a sequence of such $\boxed{\text{S}}$ or $\boxed{\text{R}}$ nodes, a *shift* operation simply pushes an additional $\boxed{\text{S}}$ node on top of the stack:

$$\boxed{\ldots \mid c_2 \mid c_1} \rightarrow \boxed{\ldots \mid c_2 \mid c_1 \mid \text{S}}$$

while a *reduce* operation of a right hand side of length n reduces the topmost n elements from the stack into a new $\boxed{\text{R}}$ node:



In particular, a reduce for a RHS with a null length pushes a childless $\boxed{\text{R}}$ node onto the LR stack:

$$\boxed{\ldots \mid c_2 \mid c_1} \rightarrow \boxed{\ldots \mid c_2 \mid c_1 \mid \text{R}}$$

Reducing θ, as described above, is not of this degenerated case, since by definition, it has a non-zero length.

Since θ is reduced at least once between two occurrences of a configuration which occurs an infinite number of time, and since reducing θ adds a $\boxed{\text{R}}$ node in our bush, we can deduce that our bush – not the LR stack – will grow indefinitely in terms of number of nodes.

Since the number of entries in the bush is the size of our LR stack and is bounded by hypothesis, for every $k \in \mathbb{N}$, there will be a stage where the bush will contain a tree of at least k nodes.

Since the number of children of any node in any tree of our bush is bounded by the length of the longest RHS in our grammar, for every $d \in \mathbb{N}$ there will be a stage where the bush will contain a tree with a depth of at least d, thereby defining a chain of d $\boxed{\text{R}}$ nodes since $\boxed{\text{S}}$ nodes have no children. Let $R_1, R_2, \ldots, R_d$ be such a path of reductions in my tree, R_1 being at the top level, that is, in the LR stack. We also define $r_1, r_2, \ldots, r_d$ as the corresponding enclosing rules (the R's are right hand sides).

We have:

$$\forall\, i > j \in \mathbb{N} \bullet R_i.LexWeight \leq R_j.LexWeight$$

since we have:

$$\exists\, \mu, \gamma \in (N \cup T)* \bullet r_j \stackrel{+}{\Rightarrow} \mu\, r_i \gamma$$

by construction of our tree.

$R_1.LexWeight$ is bounded by the total number of shift operations that have occurred before entering the cycle. Let χ be this number.

For every $j \in \mathbb{N}$, we can build a sequence R_i such that it contains a sub-sequence $R_m, R_{m+1}, \ldots R_{m+j-1}$ of j reductions such that

$$R_m.LexWeight = R_{m+1}.LexWeight = \ldots = R_{m+j-1}.LexWeight$$

by taking $d > \chi \times j$. If we have a reduction of the form:

$$\begin{tikzcd}[column sep=small] & & & \boxed{R_i} \arrow[dlll] \arrow[dll] \arrow[d] \arrow[drr] \arrow[drrr] & & & \\ c_1 & \ldots & & \boxed{c_k = R_{i+1}} & & \ldots & c_n \end{tikzcd}$$

where $R_i.LexWeight = R_{i+1}.LexWeight$, we must have:

$$\forall\, i \in 1..n \mid i \neq k \bullet c_i.LexWeight = 0$$

and thus:

$$r_i \overset{+}{\Rightarrow} r_{i+1}$$

Since the total number of rules in the grammar is bounded, by choosing j greater than this number of rules, we will have:

$$\exists\, x, y \in m..m+j-1 \bullet x \neq y \wedge r_x = r_y$$

and since $R_x.LexWeight = R_y.LexWeight$, and by transitivity of $\overset{+}{\Rightarrow}$:

$$r_i \overset{+}{\Rightarrow} r_i$$

which qualifies a grammatical cycle.

If a parser generated by `DURA` cycles with a bounded LR stack, the grammar contains a grammatical cycle, that is, a grammar rule a such that $a \overset{+}{\Rightarrow} a$.

5.5.4 The LR stack is growing

Definition 4
A void sequence is a slice of the LR stack which only contains $ReduceElements$ with $LexWeight = 0$.

Lemma 11
In a cycling parser with an ever growing LR stack, for every $n \in \mathbb{N}$, there is a stage where the LR stack contains a void sequence made of at least n elements.

At some stage, σ does not evolve any longer (See Lemma 7). Let σ' be the value of σ at this stage. Then, it is enough to wait for the LR stack to grow larger than $(\sigma' + 1) \times n$ elements to be sure that there is a void sequence of at least n elements.

Lemma 12
At any stage, the attribute $Syms$ as defined by the $Parser$ class is a prefix for a valid derivation of the root grammar rule.

This is an essential property of the LR stack, as defined formally in [29] and , and is proved easily by induction from the semantic nature of shift and reduce operations. □

From Lemma 11 and Lemma 12, one can deduce that for any positive number n, the root grammar rule supports a derivation with at least n consecutive nullable non-terminals. Since n is unbounded, these strings of nullable non-terminals cannot be produced by a finite number of non-recursive grammar rules. Since the number of grammar rules is finite, there must be at least a recursive grammar rule, that is, a non-terminal a such that $a \stackrel{+}{\Rightarrow} \mu\, a\, \lambda$. The derivations of a can then contain infinite sequences of μs or λs.

If we have $\mu \stackrel{+}{\not\Rightarrow} \epsilon$, and $\lambda \stackrel{+}{\not\Rightarrow} \epsilon$, we cannot explain an unbounded void sequence on the LR stack. At least, one of μ and λ must be reducible to an empty string.

If we have $\mu \stackrel{+}{\not\Rightarrow} \epsilon$ and $\lambda \stackrel{+}{\Rightarrow} \epsilon$, we cannot have an unbounded sequence of λ's unless we have a preceding matching infinite sequence of μs, which is of course impossible if the input is finite in size.

Hence, the only case that can lead to a cycling parsers with an unbounded LR stack is when we have $\mu \stackrel{+}{\Rightarrow} \epsilon$.

The more specific case where we have $\mu \stackrel{+}{\Rightarrow} \epsilon$ and $\lambda \stackrel{*}{\Rightarrow} \epsilon$ is a grammatical cycle $a \stackrel{+}{\Rightarrow} a$, which is already identified as a possible cause for a DURA-generated parser to cycle.

If a parser generated by DURA cycles with a growing LR stack, the grammar contains a flattened rule a such that $a \stackrel{+}{\Rightarrow} \mu\, a\, \lambda$, where $\mu \neq \epsilon$ and $\mu \stackrel{+}{\Rightarrow} \epsilon$.

This result is not entirely surprising: it applies to Tomita (even if overlooked in the original paper [66]) and to recursive ascent [77] parsers, and it follows naturally from the tight relationship between DURA and Tomita parsers (See 5.3.3 in page 237).

This configuration is referred to as *hidden left-recursive grammar rules* by Nederhof in [78].

This possible failure of Tomita's algorithm was first shown by Nozohoor-Farshi [76], together with a solution *applied directly to the intrinsic behaviour of the algorithm*. Other contributions in this area include Nerderhof and Sarbo [79], and Rekkers [80], which all have to do with the alterations to the semantic of the shared LR stacks and automaton.

These solutions were also justified by the observation by Nerderhof and Sarbo [79] that a production-level grammar for Dutch used at the Delft University of Technology included such hidden left recursions[4].

So far, we have not encountered any real-world programming language for which the grammar includes a hidden left-recursive grammar rule, but if it ever happens,

[4]This digression about existing references related to hidden left-recursive grammar rules is vastly inspided by the section on Tomita parsing in *Parsing Schematas* [36] by Klaas Sikkel.

several actions can be taken:

- Cancel the generation of the parser with a comprehensive error message. It is the developer's responsibility to adapt the grammar, just as he (or she) would to remove grammatical cycles if any before submitting it again to `DURA`.
- Consider the options listed above that have been developed to allow Tomita parsers to support hidden left-recursions, and adapt one of them to `DURA`.
- Prevent hidden left-recursions, by performing a grammar transformation described as follows: for every nullable non-terminal A, define A' as its non-nullable incarnation, that is:

  $$\begin{array}{rcl} A & ::= & A' \\ & | & \epsilon \end{array}$$

  If there is no such A', that is, if A is always null, just remove all references to A from the entire grammar. Otherwise, replace all rules of the form:

  $$B \;::=\; A\,\mu$$

  by the couple

  $$\begin{array}{rcl} B & ::= & A'\,\mu \\ & | & \mu \end{array}$$

  After having performed this substitution operation on the entire grammar, there cannot be any right hand side with a leading nullable non-terminal reference, thereby making hidden left-recursions totally impossible.

  This transformation can inflate the grammar significantly, but clearly preserves the language recognized by the parser. It also makes the parser faster, by removing the need for empty reductions when recognizing the start of some of the right hand sides.

The current version of `DURA` does not detect or remedy hidden left-recursion. If it ever proves useful to do something about this, the first of these options will probably be preferred.

5.5.5 Concluding remark

We have proved that a non-terminating `DURA`-generated parser implies one of two grammar configurations. We have not proved that these configurations *always* cause the parser to cycle. If we ensure that `DURA` does not produce a parser when facing one of these configurations, we might as well restrict ourselves excessively, and reject perfectly acceptable grammars.

In practice, this does not seem to be the case. It is intuitively clear that both of these configurations *do* cause the parser to cycle when given a not so cooperative input[5]:

[5]In theory, these configurations can be harmless if they are not reachable from the root grammar rule, but since `DURA` actually builds the grammar as the set or such reachable rules, we don't have to take this degenerated case into account.

- If we have $a \stackrel{+}{\Rightarrow} a$, backtracking on an already reduced a can always reduce it again onto itself, and cause the parser to start going forward again without having changed anything about the current state.

- If we have $a \stackrel{+}{\Rightarrow} \mu\ a\ \lambda$ with $\mu \neq \epsilon$ and $\mu \stackrel{+}{\Rightarrow} \epsilon$, when backtracking after having parsed an a, the parser will restart after parsing $\mu\ a$, then $\mu^2\ a$, $\mu^3\ a$, etc.

All in all, this proof serves its basic purpose, which was to ensure that the parser could *not* cycle unless one of these configurations was present in the grammar.

Chapter 6

DURA user guide

This section describes how to use DURA from a user point of view. Its theoretical foundations can be found starting in page 239. The reader is supposed to have at least some basic knowledge of the YAFL [31] programming language, since the interaction between DURA and YAFL is described extensively here.

6.1 Basic principles

DURA extends the YAFL programming language, by allowing the user to provide grammatical information within plain YAFL classes. Non-terminals are mapped onto classes: each non-terminal is implemented as a YAFL class. There are several practical reasons for doing so, but the most obvious justification is highly theoretical: grammar rules can be represented quite conveniently using OO constructions, and more specifically, inheritance.

When a YAFL class A that represents a non-terminal inherits from another YAFL class B that also represents a non-terminal, an implicit production $B ::= A$ is added to the grammar. For instance, when declaring an $IfStatement$ class as inheriting from the $Statement$ class, one implicitely adds a rule $Statement ::= IfStatement$ to the grammar. This feature is referred to as *reverse inheritance*. It is called "reverse" inheritance because the base class grammar rules "inherits" from the derived class grammar rule.

DURA-generated parsers build the parse tree automatically while recognizing the input. Hence, such parsers are very convenient when dealing with true compilers, that require such parse trees, and might prove inadequate for other purposes, such as command line interpreters, where one would rather perform semantical actions on the fly rather than building the parse tree.

A DURA-generated parser is meant to be built on top of an existing lexical analyzer, and besides its own ability to backtrack, it supports backtracking for the lexical analyzer as well. Typically, a DURA-generated parser can be used in conjunction with a lexical analyzer that has been built using `lexyt`.

In its current version, DURA generates parsers as YAFL source code. In future releases, we might consider generating parser in other languages as well. However, these parsers are faithfull to the OO principles. All predefined behaviours can be overridden by user-written code. For instance, after recognizing a grammar rule, the corresponding object is created, and initialized using the elements that have been found in the parsed source. A user-redefinable method (See method *DuraBase.BaseNonTerminal.Commit* on page 347) is then called, and by redefining it, semantical checks can be performed. If some of these checks fail, the parser can be instructed to cancel the reduction and to use backtracking in order to attempt to reduce another non-terminal.

Mapping non-terminals onto YAFL classes has one interesting side-effect: it gives the ability to define *modular* grammars, closely following modularity as supported by YAFL. This ability has drawbacks that are described hereafter, but it enables the development and eases the maintenance of large parsers.

6.2 The mechanism

DURA uses two passes to generate a parser:

1. First, all modules are compiled using the DURA compiler, which is nothing else than an improved YAFL compiler (See figure 6.1, first part). In addition to the ability to compile grammatical information, this compiler can compile any source code that would be accepted by the plain YAFL compiler[1]. When the DURA compiler compiles a module, besides the normal output of the plain YAFL compiler, it also generates an intermediate file with the "$.cy$" extension that contains the grammatical information in a synthetic form. Hence, there is one such "$.cy$" file for every module.

2. Then, all these intermediate files are read again, and the corresponding parser is generated (See figure 6.1, second part). Depending on the options specified on the command line (See page 274), DURA might also generate a "$.tab$" file, that contains a tabular description of the push down automaton of the parser.

This is how DURA supports modularity, in a way which is quite similar to a compiler for a modular language, that generates one object file for each source file of a program. Then, a separate linker is used to gather all these object files into an executable image.

6.2.1 Interpreted and compiled parsers

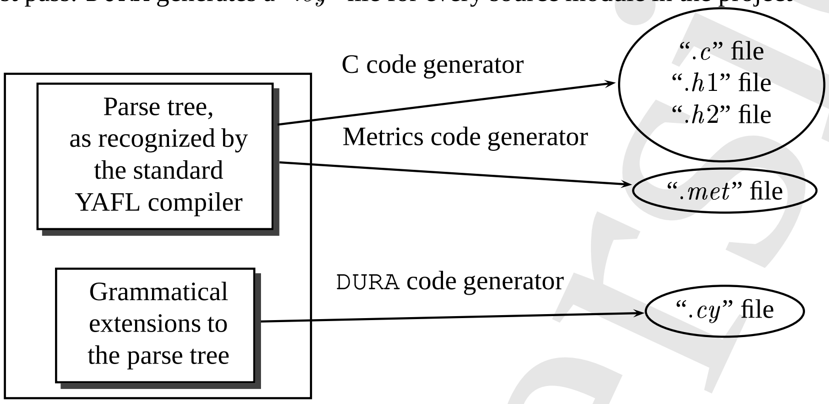
DURA can produce the transition automaton in the parser as a number of nested CASE statements to yield the best performance one can get, given the level of abstraction of the YAFL computing model. Such parsers will be referred to as *compiled parsers*.

The YAFL source files produced for compiled parsers grow fast as the grammars increase in size, up to a point where actually compiling the parser becomes a performance issue, or even, a logistics issue when the YAFL compiler simply cannot handle a source file of such size.

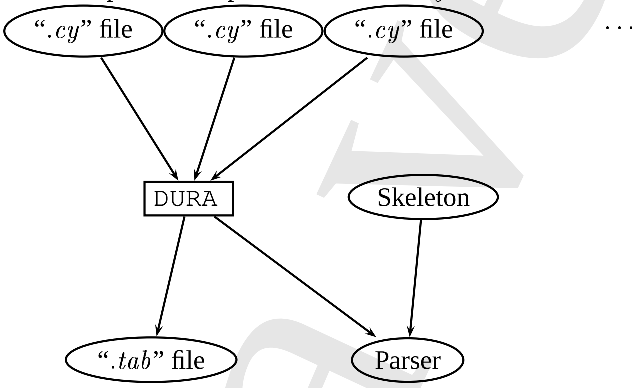
[1] It is perhaps more accurate to state that the DURA compiler can compile plain YAFL code, but it also accepts *grammatical extensions*.

Figure 6.1 The two passes of `DURA`

First pass: `DURA` generates a "*.cy*" file for every source module in the project



Second pass: `DURA` produces a parser out of all "*.cy*" files



`DURA` can also produce the transition machinery in an external file, referred to as the "*.tab*" file. This file contains a description of the push-down automaton, and is read in runtime by the parser. Such *interpreted parsers* are of course slower than their compiled counterparts but allow the generation of parsers even for abnormally large grammars.

At the day of this writing, all the parsers we generate are interpreted parsers, up to a point where compiled parsers cannot be generated any longer because the support for some of the latest addition to `DURA` *– such as error recovery – has not been implemented yet for compiled parsers. Depending on how* `DURA` *is used in the future, we might complete the support for compiled parsers, or remove it entirely.*

6.3 Non-terminal classes

A class that is intended to be used as non-terminal must fulfill several conditions:

1. It must be public so that the parser can even be aware of it, and CREATE instances of this class. If the class was private, the parser, generated in an external module, would not have access to it, and would not be able to create instances for it.

2. Similarly, it must support a parameterless CREATE method, so that the DURA-generated parser can CREATE it without having to figure out parameters out of the blue. However, this CREATE method can be redefined or inherited, as long as it is parameterless.

3. Be parameterless. One might wonder what the true meaning of a parameterized non-terminal might be.

4. Inherit from the BaseNonTerminal class (See page 346), so that generic operations that must be applicable to any non-terminal can be used.

5. Be marked by the NONTERMINAL keyword, consistently between definition and implementation. In other words, a class marked as NONTERMINAL in an implementation module must also be marked as NONTERMINAL in the definition (which exists since all such classes must be public, see above).

6.4 Acceptable terminals

The set of terminals that can be used within grammar rules is defined by a module which name is given as command-line parameter when invoking DURA (see page 274).

The definition module must include a constant definition for each of the acceptable terminals. For instance, for all the examples listed in this user guide, we'll assume that a definition module such as depicted in figure 6.2 has been indicated using the appropriate command-line switch. Such a definition module is generated automatically using lexyt (See chapter 3).

6.5 Using terminals and non-terminals

Within the implementation of a class that has been marked as NONTERMINAL, a grammar clause can be specified before any declaration (method, constant, attribute) by using the keyword GRAMMAR.

Let's first consider that a grammar clause is made of a sequence of terminals and non-terminals, as in figure 6.3.

In this example, LParen, RParen, QuestionMark and Colon denote terminals, that were defined in a lexical specification. When compiling these grammar rules, this lexical specification must have been given to DURA for reference.

We also assume that Expression is a non-terminal class visible the current scope, imported or declared within the same module.

This example basically claims that a ConditionalExpression is made of three sub-expressions, separated by a question mark and a colon, within parentheses.

A grammar clause can also be made of several *grammar rules* that are related by an implicit disjunction, and separated by semicolons in the source file, as in figure 6.4

Figure 6.2 Definition module as generated by `lexyt`

```
DEFINITION MODULE CLex;

FROM YLexicalAnalyzer IMPORT YLexicalAnalyzer;

CLASS CLex;
  INHERITS YLexicalAnalyzer;

  REDEFINE METHOD CREATE;

  CONST
    Date = "2/1/1997";
    LexYtVersion = "3.9";
      BACKTRACKLEVEL = 3;
      DYNAMICBACKTRACKLEVEL = FALSE;
      While = FIRSTTOKEN +0;
      For = FIRSTTOKEN +1;
      Do = FIRSTTOKEN +2;
      Goto = FIRSTTOKEN +3;
      Else = FIRSTTOKEN +4;
      Extern = FIRSTTOKEN +5;
      Static = FIRSTTOKEN +6;
      Switch = FIRSTTOKEN +7;
      ...
      LASTTOKEN = FIRSTTOKEN + 59;

END CLex;
```

Figure 6.3 Simple example, using terminals and non-terminals only

```
NONTERMINAL CLASS ConditionalExpression;
  INHERITS BaseNonTerminal;

  GRAMMAR
    LParen Expression QuestionMark
           Expression Colon
           Expression RParen;

END Assignment;
```

which describes two forms of loops, and is the direct translation of the following grammar rule:

$$\begin{array}{lcl} \mathit{LoopStatement} & ::= & \mathit{While\ Expression\ Do\ Statement}\ \mid \\ & & \mathit{Do\ Statement\ While\ Expression} \end{array}$$

6.6 Extended operators

Using simple grammar rules as described above, one can express any context-free grammatical construction. This is exactly the level of functionality provided by `yacc`. However, in practice, building a parser for a reasonably complex grammar using these forms of grammar rules only can prove very tedious.

This section describes extended operators, that can be used to express complex

Figure 6.4 Implicit disjunction in grammar clauses

```
NONTERMINAL CLASS LoopStatement;
  INHERITS BaseNonTerminal;

  GRAMMAR
    While Expression Do Statement ;
    Do Statement While Expression ;

END LoopStatement;
```

operations in a more convenient and expressive way. More specifically, such operators will remove the need for phony non-terminal classes, that are required to express grammatical constructs such as repetition, but serve no purpose at the application level.

6.6.1 Disjunction and conditionals

Figure 6.5 Simple example of disjunction

```
NONTERMNAL CLASS Expression;
  INHERITS BaseNonTerminal;

  GRAMMAR
    Expression (Plus OR Minus) Term;
...
```

The example depicted in figure 6.5 shows that a grammatical expression can be made of an `Expression`, followed by a `Plus` or `Minus` token, followed by a `Term`. Grammatically, this example is equivalent to the example presented in figure 6.6, where the operator is analyzed by a separate nonterminal. The phony class `ExpressionOp` serves no practical purpose, and makes the grammar larger and harder to maintain.

Similarly, the conditional operator "?" can be used to indicate that a given grammatical construct is optional, that is, that the rule is acceptable with or without that grammatical construct.

Figure 6.6 Expanded grammar rules without explicit disjunction

```
NONTERMINAL CLASS Expression;
  INHERITS BaseNonTerminal;

  GRAMMAR
    Expression ExpressionOp Term;
...
END Expression;

NONTERMINAL CLASS ExpressionOp;
  INHERITS BaseNonTerminal;

  GRAMMAR
    Plus;
    Minus;
...
END ExpressionOp;
```

Figure 6.7 Conditional operator

```
IMPLEMENTATION MODULE Statements;

FROM BaseNonTerminal IMPORT BaseNonTerminal;
FROM Expressions IMPORT Expression;

  NONTERMINAL CLASS IfStatement;
    INHERITS BaseNonTerminal;

    GRAMMAR
      If LParen Expression RParen
         Statement
  (Else Statement)?;

  END IfStatement;

END Statements;
```

In the example depicted in figure 6.7, the `ELSE` part of the `IfStatement` is optional. The `Else` keyword and the reference to the `Statement` non-terminal are grouped using parentheses, so that the optional postfix operator "?" applies to both of them together. Otherwise, it applies to the closest component in the grammar clause.

6.6.2 Grammatical `CASE` operator

When a non-terminal, or more generally, a fragment in a grammar rule, can have several mutually exclusive forms, an implicit (using several grammar rules within a clause, as shown in figure 6.4) or explicit (using the OR operator) disjunction can be used, as in figure 6.8.

Figure 6.8 Brute force disjunction

```
NONTERMINAL CLASS Assignment;
  INHERITS BaseNonTerminal;

  GRAMMAR
    Move Expression To Target,
    Expression Arrow Target,
    Target Becomes Expression;

END Assignment;
```

However, in such cases, one might want to know which of the forms has been recognized. This can be achieved by storing each of these forms in separate non-terminal classes with different names, and then use WHAT statements to test the class of a recognized non-terminal.

DURA supports a more convenient notation for this purpose, that does not require the addition of a large number of otherwise useless classes.

Figure 6.9 `CASE` operator

```
NONTERMINAL CLASS Assignment;
  INHERITS BaseNonTerminal;

  GRAMMAR
    CASE TheForm OF
      CobolForm:
        Move Expression To Target
        END;
      ArrowForm:
        Expression Arrow Target
        END;
      PascalForm:
        Target Becomes Expression
        END;
      END;

  CONST
    CobolForm= 997;
    ArrowForm = 998;
    PascalForm = 999;

  VAR
    TheForm: INTEGER;

END Assignment;
```

A `CASE` expression as supported only within grammar rules is made of several components:

- An integer attribute
- A non-empty set of alternatives. Each of these alternatives is itself made of two sub-components: a constant integer expression, and a grammar expression.

In terms of grammar, a `CASE` expression is a disjunction of the grammar expressions of its alternatives. When one of the alternatives has been recognized, the attribute attached to the `CASE` expression is given the value attached to the alternative.

For instance, after recognizing an instance of the `Assignment` class presented in figure 6.9, if the second form (that is, `Expression Arrow Target`) is recognized, the attribute `TheForm` holds the value 998 (the value attached to the `ArrowForm` constant).

6.6.3 Repetition

Repetition is commonly represented using recursion in grammar rules. `DURA` supports repetition using an ad hoc notation, and produces intermediate recursive grammar rules if required when generating the transition tables for the parser.

DURA supports two postfix operators for repetition:

- "+" to denote one or more repetitions of the operand
- "*" to denote zero or more repetitions of the operand

Besides, DURA also supports a BY clause, that indicates a separator. For instance, in the example shown in figure 6.10, `Expression * BY Comma` will match:

$$
\begin{array}{lcl}
Expr_* & ::= & \epsilon \\
& | & Expression \\
& | & Expression\ Comma\ Expression \\
& | & Expression\ Comma\ Expression\ Comma\ Expression \\
& \ldots &
\end{array}
$$

Figure 6.10 The repetition operator

```
NONTERMINAL CLASS FunctionCall;
  INHERITS BaseNonTerminal;
  GRAMMAR
    Ident LParen Expression * BY Comma RParen

END FunctionCall;
```

The separator will often be a simple token reference, but it can be an arbitrarily complex grammar expression.

6.6.4 Inheritance, of course

Besides explicit references within grammar rules, DURA supports another way of connecting non-terminal classes, by using *reverse inheritance*. When a nonterminal class D inherits from a base nonterminal class B, DURA automatically considers that an implicit grammar rule $B\ ::=\ A$ is added to the grammar.

The example listed in figure 6.11 declares a `Statement` base class, as well as several derived classes. Whenever a `Statement` (grammatically speaking) is expected, any grammar form of one of `Statement`'s derived classes can be provided. If an attribute is supposed to refer to the `Statement` that has been recognized, it might refer to an instance of any of `Statement`'s derived classes, so that everything is typewise correct.

6.7 Terminal sets

When a transition can be made for a number of terminals, one can use an explicit disjunction, as in

```
Expression (Plus OR Minus) Expression
```

Figure 6.11 Reverse inheritance

```
IMPLEMENTATION MODULE Statements;

FROM BaseNonTerminal IMPORT BaseNonTerminal;

  NONTERMINAL CLASS Statement;
    INHERITS BaseNonTerminal;
  END Statement;

  NONTERMINAL CLASS IfStatement;
    INHERITS Statement;
...
  END IfStatement;

  NONTERMINAL CLASS WhileStatement;
    INHERITS Statement;
...
  END WhileStatement;

  NONTERMINAL CLASS AssignmentStatement;
    INHERITS Statement;
...
  END AssignmentStatement;

END Statements;
```

However, this scheme has two drawbacks:

- It soon becomes cumbersome to list large numbers of such terminals explicitely if the set grows beyond half a dozen elements
- When processed by `DURA`, such a construction produces an intermediate grammar rule, which must be reduced, possibly undone, etc. This construction induces a possibly significant performance penalty.

To better cope with these issues, `DURA` also supports an explicit notations to describe transitions on sets of terminals. These transitions do not require an additional reduction, thereby providing an extra level of abstraction without any performance penalty.

These sets are specified between square brackets, as in

```
Expression [Plus, Minus] Expression
```

Besides, one can use the `TO` keyword, to define ranges of terminals, as in

```
[Array TO While]
```

which would match all the reserved words defined in a typical programming language. This range is based on the order in which the lexemes are defined in the lexical specification.

One can also use parentheses, to group such terminal set expressions together, and the “-” operator to compute set differences, as in

```
[(Array TO While) - Procedure]
```

Last but not least, “*” denotes the set of all possible terminals.

All these constructions can be used together, as in the (admittedly, useless) example depicted in figure 6.12.

Figure 6.12 Terminal set transitions

```
GRAMMAR
  Begin [ * - (LBrace, RBrace,
               Array TO While)] End;
```

Terminal sets as described here can appear as redundant, since the same effect can be obtained using the `OR` operator: `[Plus, Minus]` is equivalent to `Plus OR Minus`. However, they are very different in nature, and serve different purposes:

- Terminal sets only contain terminals, while the `OR` operator can be used to build disjunctions of arbitrary grammatical expressions.
- Terminal sets are more efficient than the corresponding `OR`-based expression, and support high-level set-based operators to build the terminal set.

6.7.1 Associativity and precedence

Expression arithmetic grammar commonly includes factors and terms to distinguish between the different precedences of operators. Similarly, explicit recursion is used to specify associativity.

$$
\begin{array}{lcl}
E & ::= & E + T \mid \\
 & & T \\
T & ::= & T * F \mid \\
 & & F \\
F & ::= & Number \mid \\
 & & (E)
\end{array}
$$

This situation gets awkward when the grammar must support many different levels of precedence, in which case many intermediate classes should be used just to implement precedence and associativity. DURA allows the explicit marking of associativity and precedence level on grammar rules that are left- and right-recursive[2] using the

Figure 6.13 Associativity and precedence

```
NONTERMINAL CLASS BinaryExpression;
  INHERITS Expression;

  GRAMMAR
    Expression [Plus, Minus] Expression :
                         LEFTASSOC PRECEDENCE 5;
    Expression [Star, Slash] Expression :
                         LEFTASSOC PRECEDENCE 3;
    Expression Power Expression :
                         RIGHTASSOC PRECEDENCE 1;

END BinaryExpression;
```

`LEFTASSOC`, `RIGHTASSOC` and `PRECEDENCE` keywords after every grammar rule within a clause.

For instance, `BinaryExpression` in figure 6.13 is right- and left-recursive, since a rule $Expression\ ::=\ BinaryExpression$ is added to the grammar by reverse inheritance.

The actual values for the precedence only have a relative meaning, indicating that one rule with a highest precedence should be considered before another one with a lower precedence. These precedence levels are system-wide values, and can be used to remove ambiguities between grammar rules from different modules.

6.8 Assignment

Recognizing a grammar expression is valuable. However, it is only part of the story. One might expect the parser to actually use the subparts of a grammar expression, and perform significant actions on these subparts.

`DURA` supports automatic building of a parse tree by assignment. On every grammar expression (or sub-expression), one can attach a *target* which must be an identifier that refers to a non-once attribute within the current class. Whenever a new instance of this class will be created by parsing a grammatically correct fragment of an input stream, the attributes within the newly created instance will be assigned according to targets specified in the grammar expression[3].

Depending on the target's type, that is, the type attached to the attribute that is used as target, the assignment can take entirely different forms:

- If the target is an array of character, it is given the image of the rightmost token used to recognize the sub-expression assigned to the target. If the sub-expression

[2]This restriction is simply dictated by the fact that associativity and precedence only make sense if the grammar is left- and right-recursive. Associativity is required to remove an ambiguity which is only present if the grammar rules are recursive.

[3]This all sounds very complex, but it is amazingly simple to use in practice. Please read further, and have a look at examples in figures 6.14, 6.15, 6.16, 6.17 and 6.18 before rejecting this as too awkward to use.

reduces to an empty string, the target receives `VOID`

- If the target is an integer, the code of the rightmost token used to recognize the sub-expression assigned to the target. If the sub-expression is empty, the target receives 0.
- If it is a boolean, it is given a value that indicates whether a given grammar expression is void or not
- If the target is a list, it can receive a sequence of sub-expressions built using the "*" and "+" operators (See figure 6.18 in page 271).
- If the target is a non-terminal class attribute, it receives the sub-tree produced by the parsing of the sub-rule.

One can have several assignments within a single grammar clause, as in:

```
GRAMMAR
  If TheCond := Expression Then TheBody := Statement;
```

but multiple assignments of a single sub-expression is not supported:

```
GRAMMAR
  Code := (Str := [Plus, Minus])
```

The assignment operation has a higher precedence than the implicit concatenation in `DURA`'s grammar expression. Hence, when writing `A Attr:= B C D`, the assignment only applies to `B`. To assign the entire rule postfix, one must write `A Attr:= (B C D)`.

6.8.1 Assigning a lexeme

When the target of an assignment is an integer attribute, it will be given the code of the last token recognized as part of the attached grammar expression, or 0 if the corresponding expression is empty.

Figure 6.14 Assigning a lexeme

```
NONTERMINAL CLASS BinaryExpression;
  INHERITS BaseNonTerminal;

  GRAMMAR
    Expression Op := [Plus, Minus] Expression;

  VAR
    Op: INTEGER;

END BinaryExpression;
```

In the example depicted in figure 6.14, the attribute `Op` will receive the value `Plus` if the token is a "+", and the value `Minus` is the token is a "-". The values for `Plus` and `Minus` come from the lexer module that must have been specified on the command line when invoking `DURA` (See figure 6.2 on page 259).

6.8.2 Assigning an indicator variable

One can also assign a boolean variable, that will receive the value `TRUE` if its corresponding expression is not void, and `FALSE` otherwise.

Figure 6.15 Assigning an indicator (BOOLEAN) variable

```
NONTERMINAL CLASS FunctionDecl;
  INHERITS BaseNonTerminal;

GRAMMAR
  StaticFlag := Static ? Ident Formals SemiColon;

VAR
  StaticFlag: BOOLEAN;
...
END FunctionDecl;
```

In the example depicted in figure 6.15, the boolean attribute `StaticFlag` receives the value `TRUE` if the corresponding expression `Static ?` is not empty, that is, if the keyword `static` has been recognized. In all other cases, `StaticFlag` is set to `FALSE`.

6.8.3 Assigning a lexeme image

If the target is an attribute which type is array of character, it will receive the textual value of the last lexeme which is part of the sub-expression, that is, the sequence of characters that have been recognized as representing a lexeme.

This is the only kind of array that can be used as target for an assignment within a grammar expression.

In the example depicted in figure 6.16, `FuncName` will receive the image of the function name, that is, the character string as it was found originally in the source code.

6.8.4 Assigning a non-terminal

The most common and most useful form of assignment within a grammar rule assigns a non-terminal to an attribute which class matches the non-terminal's class.

In the example depicted in figure 6.17, after parsing, `TheExpr` will receive the `Expression` instance that has been created based on the sequence of tokens that have been recognized as matching the grammar rules for `Expression`. More specifically, `TheStmt` will receive a `Statement` instance, or any `Statement`-derived class

Figure 6.16 Assigning a lexeme image

```
NONTERMINAL CLASS FunctionDecl;
  INHERITS BaseNonTerminal;

GRAMMAR
  StaticFlag := Static ? FuncName := Ident
    Formals SemiColon;

VAR
  StaticFlag: BOOLEAN;
  FuncName: ARRAY OF CHAR;
...
END FunctionDecl;
```

Figure 6.17 Assigning a non-terminal

```
NONTERMINAL CLASS IfStatement;
  INHERITS Statement;  -- Assuming that Statement
                       -- inherits from
                       -- BaseNonTerminal
  GRAMMAR
    If LParen TheExpr := Expression RParen
       TheStmt := Statement
     (Else
       TheElseStmt := Statement)? ;

  VAR
    TheExpr: Expression;
    TheStmt, TheElseStmt: Statement;

END IfStatement;
```

instance, such as `IfStatement`, using the reverse inheritance feature described in page 264.

6.8.5 Assigning a list

One can also assign a compound non-terminal using repetition operators “+” and “*” into an attribute. This attribute’s type `NonTerminalList` (See page 360) or one of its derived classes. In addition, three conditions must be fulfilled:

- The type’s class must be public
- If the class is parameterized, the target attribute’s type must be constrained. For instance, an attribute with type `NonTerminalList` cannot be used. It must be constrained, for instance as a `NonTerminalList(Statement)`.
- All the parameters must denote public classes

Figure 6.18 Assigning a list

```
NONTERMINAL CLASS FunctionCall;
  INHERITS Expression;

  GRAMMAR
    FuncName := Ident
    LParen
    Args := (Expression * BY Comma)
    RParen ;

  VAR
    FuncName: ARRAY OF CHAR;
    Args: NonTerminalList(Expression);

END FunctionCall;
```

In the example depicted in figure 6.18, `Args` will receive the list of arguments of the function call. However, if no argument is found at all (in case of something such as "`f()`"), `Args` will be `VOID`, instead of denoting an initialized list with no element.

6.8.6 Implicit assignment

When a target's type is a non-terminal class and it is assigned an expression of that very same non-terminal class, it is sufficient to just indicate the attribute name, without specifiying the assignment from the sub-expression onto the attribute explicitely.

6.8.7 Assignment to `THIS`

`DURA` also supports a special case of assignment, that does not actually assign an attribute, but which, given a rule, replaces it by a subrule which must be of the same type or derived.

For instance, consider the example depicted in figure 6.20. It assumes, of course, that `Expression` supports derived classes with unary and binary operators, function calls, literals, etc. It indicates that `LParen Expression RParent` is a possible production for `Expression`, and that when it is reduced, there is no need to keep track of the parentheses, the enclosed `Expression` can be returned directly. In other words, the enclosed `Expression` *replaces* the enclosing one with parentheses.

It is an ad hoc solution for a problem that is encountered at least once in virtually every grammar, namely, syntactical recursion with no semantic added value.

6.9 Pragmas

This section describes the various pragmas that are recognized by `DURA` to tune the behaviour of its generated parsers. These pragmas are consistent with the common

Figure 6.19 Implicit assignment

```
NONTERMINAL CLASS IfStatement;
  INHERITS Statement;

  GRAMMAR
    If LParen
       TheExpression := Expression
RParen
StatementList;

  VAR
    TheExpression: Expression;

END IfStatement;
```

This example can be rewritten more conveniently as:

```
NONTERMINAL CLASS IfStatement;
  INHERITS Statement;

  GRAMMAR
    If LParen TheExpression RParen StatementList;

  VAR
    TheExpression: Expression;

END IfStatement;
```

syntax defined for plain YAFL pragmas [31]. If encountered by a plain YAFL compiler that does not recognize them in any way, a warning is issued and they are ignored.

The other extensions to the plain YAFL compiler – namely, the recognition of the `NONTERMINAL` keyword and the grammar expressions – will not be recognized at all by a plain YAFL compiler, and will produce a reasonably comprehensive error message.

6.9.1 AutomaticRecovery

The `AutomaticRecovery` pragma must be used without arguments within a non-terminal class definition or implementation. Its presence indicates that `DURA` must handle error-recovery automatically for this non-terminal.

When dealing with a backtracking parser such as generated by `DURA`, error-recovery has a slightly different meaning from what it might mean with more conventional parsers. Error recovery is only fired when all possible alternatives compatible with snips (See page 273) have been tested, while a more conventional parser would start error-recovery as soon as it is in a state where no valid transition can be performed.

`DURA`'s error-recovery algorithm can be described as follow:

- Whenever the parser's backtracking algorithm fails – that is, whenever it has tried

Figure 6.20 Assignment to `THIS`

```
NONTERMINAL CLASS Expression;
  INHERITS BaseNonTerminal;

  GRAMMAR
    LParen THIS := Expression RParen;

END Expression;
```

all the possible ways of interpreting the input, and has found that there is no way the various grammar rules can be combined in a way that matches the input – the system restores its *most advanced state*, that is, the state it was in when analyzing the most advanced token it has read[4]. Since the parser has failed, we assume that the input is somehow incorrect. At least a single token must be skipped before a successfull recovery can take place.

- The parser starts skipping the tokens following this most advanced state, testing successively all their possible interpretation (using lexical backtracking)
- For each of these tokens, it checks the entire LR stack for partially evaluated non-terminals – that is, non-terminals for which recognition is currently active but not terminated yet. In other words, only grammar rules that are currently visible from the LR stack and which are not reduced yet will be taken into account.
- If one of these non-terminals is marked using the `AutomaticRecovery` pragma, and if the current token on the input is in that non-terminal's `FOLLOW`[5] set, that non-terminal is reduced artificially[6], and the system continues.
- If no such non-terminal can be found on the LR stack, the next token is read from the input, to see whether it can possibly *follow* one of the recoverable non-terminals that the system is currently trying to reduce.

In practice, the `AutomaticRecovery` pragma is used in a way quite similar to `yacc`'s error recovery system. One basically marks essential building blocks of a grammar such as statements, blocks, procedures or modules with this pragma to help the parser synchronize after an error has occurred in resp. a statement, a block, a procedure or a module.

6.9.2 Cut and Snip

If uncontrolled, backtracking can yield exponential execution times. Therefore, while it is clear that backtracking is a major asset for `DURA`, we must provide ways to control

[4]This most advanced state is named the *sting* in the generated parser skeleton (See *Parser skeleton.Parser skeleton.TheSting* on page 870).

[5]The `FOLLOW` set attached to a non-terminal N is the set of terminals that can follow N in a valid derivation of the root grammar rule.

[6]By artificially, one understands that this reduction is forced. It could not have occurred within normal circumstances, since some part of the corresponding grammar rule cannot be found

it, and avoid the disastrous effects of unbounded backtracking.

To achieve this purpose, one can assign a limiting clause to a grammar clause or to an entire grammar rule. Such limiting clauses apply to already parsed components, when a parser attempts to undo them to find another valid alternative. This clause has no effect on the grammar rule in forward mode, that is, while synthetizing the rule in normal, left to right fashion. The limiting clauses enters the scene when the parser fails *after* having reduced a grammar rule, when considering the possibility of undoing the reduction to find an alternate path which might, hopefully, lead to a successfull recognition of the input.

Without such limiting clauses, the recognition of every possible piece of input can be challenged for a possible alternate interpretation, while experience shows that while some constructs can be misunderstood, due to ambiguities in the language or awkwardnesses in the grammar, many other constructs are unambiguous. Stated otherwise, there are constructs which, when recognized, cannot lead to another valid interpretation.

The two forms of limiting clauses supported by DURA are:

SNIP that indicates that the grammar clause(s), once analyzed successfully, should not be undone to search for another valid interpretation of the input. However, what has been recognized *before* the grammar clause(s) can be challenged for another interpretation.

When backtracking encounters a snipped grammar rule, the grammar rule is ignored entirely, and backtracking resumes *before* the beginning of the grammar rule.

CUT that indicates that the grammar clause(s), once analyzed succesfully, ought to be interpreted as a checkpoint, and commits themselves as well as whatever precedes as valid. In other words, when a parser attempts to unparse a grammar clause that has been marked as CUT, it stops looking for a way to match the input with the grammar, and starts the error recovery mechanism.

These limiting clauses should be used cautiously, since they inhibit backtracking, and might, in theory, prevent perfectly valid input from being parsed. In practice, however, when dealing with real world grammars, some grammar constructions appear as basic building blocks of a language (functions, paragraphs, etc.). Such constructions are usually good candidates for *CUT* and *SNIP* limitation. When an entire function has been analyzed, there is usually no need to try another way of analyzing it, and, in some languages where functions cannot be nested such as C, there is little need to challenge the way everything that precedes a function has been parsed either.

6.10 Using the DURA compiler

6.10.1 Command line options

Besides the command line options that are recognized by the plain YAFL compiler [31], DURA recognizes options of its own, listed in table 6.1.

Table 6.1: Command-line options for DURA

-lf$Fname$	Produce the interpreter's table in $Fname$. If this option is used, DURA generates an interpreted parser (table-driven) rather than a compiled parser (based on nested CASE statements for the automaton). This option should be used when the generated parser in YAFL gets too large to be compiled conveniently (See page 256).
-ll$Module.Class$	Use class $Class$ from module $Module$ to gather the constants attached to the lexical constants used in grammar rules. Typically, this module should have been generated using `lexyt` (See figure 6.2 in page 259).
-lt$Module.Class$	Generate the aggregated parser in $Module.Class$. In other words, this is the module where DURA must generate the parser, and which must be compiled and linked with the user's application.
-lr$Module.Class$	Use $Module.Class$ as root non-terminal for the generated parser. This non-terminal serves as goal production for the parser.
-lsd$FName$	Use $FName$ as skeleton file for definition modules
-lsi$FName$	Use $FName$ as skeleton file for implementation modules
-lvnum	Set the generated parser's verbosity level to num
-lvd	The generated parser's verbosity level is dynamic
-ly$FName$	Produce a `yacc`-equivalent parser in $fname$. This parser includes the grammar in flattened form, and no semantic action at all. It is provided for troubleshooting, sizing and debugging purposes only.

6.10.2 Compiling a single DURA module

Using DURA to compile a single module (and generate the intermediate "$.cy$" grammar file in the process), or all the modules in a YAFL project, is very similar to running the plain YAFL compiler. Of all the options listed in table 6.1, only "`-ll`" must be specified, to indicate the module and class where the constants for the lexical tokens can be found.

6.10.3 Producing a DURA parser

By invoking DURA with the "`-lt`" command-line option, one does not compile modules, but gathers the synthetic grammatical information produced by the plain DURA compilations to produce a working parser. It is the explicit usage of this options that decides whether a plain compilation or a parser generation must take place.

For DURA to be able to produce a parser, the "`-p`" option (as defined in [31]) that gives a project name must be specified as well, since parser generation happens on a global (understand: project) basis rather than at the module level.

6.11 Reporting

When producing a parser, DURA also generates a log file named `dura.log`, that contains various pieces of informations that can prove useful when one tries to better understand its behaviour. `dura.log` contains the flattened grammar, that is produced automatically from the higher-level construct defined in the YAFL classes, and the push down automaton, indicating how every state and transition in the automaton relates to the flattened grammar, and to the existing terminals and non-terminals in the grammar.

For instance, given a small grammar made of the following classes:

```
  NONTERMINAL CLASS Expression;
    INHERITS BaseNonTerminal;

    GRAMMAR
      LParen THIS := Expression RParen;

  END Expression;
-----------------------
  NONTERMINAL CLASS UnaryExpr;
    INHERITS Expression;

    GRAMMAR
      Minus TheArg : RIGHTASSOC PRECEDENCE 1;

    VAR
      TheArg: Expression;

  END UnaryExpr;
-----------------------
  NONTERMINAL CLASS FunctionCall;
    INHERITS Expression;

    GRAMMAR
      FuncName := Ident
      LParen (Actuals := Expression * BY Comma) RParen;

    VAR
      Actuals: NonTerminalList(Expression);
      FuncName: ARRAY OF CHAR;

  END FunctionCall;
-----------------------
  NONTERMINAL CLASS Literal;
    INHERITS Expression;

    GRAMMAR
      Integer;
  END Literal;
-----------------------
  NONTERMINAL CLASS BinaryExpr;
    INHERITS Expression;

    GRAMMAR
      Left Op := [Plus, Minus]: LEFTASSOC PRECEDENCE 30;
      Left Op := [Star, Divide] Right: LEFTASSOC PRECEDENCE 20;
      Left Op := (Power) Right: RIGHTASSOC PRECEDENCE 10;

    VAR
      Op: INTEGER;
      Left, Right: Expression;

  END BinaryExpr;
```

Figure 6.21 shows the beginning of this log file. The first block described the `ROOT` grammar rule, indicating that it is not nullable. In a line such as:

```
2> 0/[0] (Calc.Expression:3) [1] EOF [2]<NOTNULL> Final reduction
```

the various informations that are provided must be interpreted as:

Figure 6.21 `dura.log`, the flattened rules

```
ROOT:  <NOT NULLABLE> :-
2>  0/[0] (Calc.Expression:3) [1] EOF [2]<NOTNULL>  Final reduction
FIRST:
  Minus LParen Integer Ident Comma
LAST:
  EOF

1 <NULLABLE> :-
1>  1/[3] (5) [4]<NOTNULL>
0>  2/[5]<NULLABLE>
FIRST:
  Minus LParen RParen Integer Ident Comma
LAST:
  Plus Minus RParen Integer Comma
FOLLOW:
  RParen

2 <NOT NULLABLE> :-
2>  3/[6] Minus [7]  Calc.UnaryExpr.TheArg!(2) [8]<NOTNULL>
                                     Fetch non-terminal: UnaryExpr(0)
1>  4/[9] (7) [10]<NOTNULL>
FIRST:
  Minus LParen Integer Ident Comma
LAST:
  RParen Integer Comma
FOLLOW:
  EOF Plus Minus Star Power Divide RParen
  Comma

3 (Calc.Expression) <NOT NULLABLE> :-
2>  5/[11]  Calc.BinaryExpr.Left!(Calc.Expression:3) [12]
                                         Calc.BinaryExpr.Op!Plus Minus
 [13]<NOTNULL> Fetch non-terminal: BinaryExpr(0)
1>  6/[14] (6) [15]<NOTNULL>
FIRST:
  Minus LParen Integer Ident Comma
LAST:
  Plus Minus RParen Integer Comma
FOLLOW:
  EOF Plus Minus RParen Comma
...
```

- `2>` gives the length of the flattened rule expressed in number of symbols, that is, terminals or non-terminals (in this case, 2).
- `0/` is the right hand side numbers.
- `[0]`, `[1]`, etc. are item numbers.
- `(Calc.Expression:3)` denotes a reference to the grammar rule number 3, indicating that it yields an instance of a class named `Expression` defined in a module named `Calc`.
- `EOF` is a terminal, that terminates the root grammar rule of all grammars
- `<NOTNULL>` indicates that the right hand side is not nullable, and what follows is some information about the kind of reduction which is attached to this rule.
- The `FIRST`, `LAST` and `FOLLOW` sets are described verbatim.

These conventions are used to describe all the grammar rules in the flattened representation. It is followed by a description of the corresponding push-down automaton, as described in figure 6.22

As many other things in DURA, the states are numbered. A state is made of a number of *items*, which are basically positions within grammar rules. The example depicted in figure 6.22 shows the items, by indicating a right hand side (`7:14` indicates RHS number 14, that comes from rule number 7), and by indicating the position of the item within the right hand side by using a star "*".

After each state in the log files, the the various actions that can be taken by the parser when in this state are listed. Terminal transitions indicate how the parser can evolve on a *shift* operation while non-terminal transitions implement the *goto* function as defined for LR parsers in .

Finally, the various reductions that can occur in a state are listed, indicating which right hand sides can be reduced.

6.12 Base classes and skeleton file

The parsers produced by DURA make extensive use of classes or other components defined elsewhere in this document:

BaseNonTerminal (See page 346) is the base class for all non-terminals managed by DURA. It provides generic services (position identification, hierarchy management, etc.). Instance of `BaseNonTerminal` are never created directly: DURA creates instance of user-defined `BaseNonTerminal`-derived classes to support static typing as much as possible.

NonTerminalList (See page 360) serves the same unification purpose, except that it deals with collections rather than single non-terminals. This class and its derived classes can prove useful when dealing with repetitions ("+" and "*" operators).

YLexicalAnalyzer (See page 183) is the base class for the lexers that can be used in conjunction with DURA.

The skeleton file (See page 865) contains all the parts that are common to all DURA-generated parsers, as well as codes that will be replaced by grammar-specific information by DURA.

6.13 Restrictions

6.13.1 Cycles

DURA cannot produce parsers for grammars with cycles, that is, a non-terminal A that can be reduced to A. In other words, $A \stackrel{*}{\Rightarrow} A$. In practice, a grammatical cycle is likely to be the consequence of an error in the grammar.

DURA detects and reports grammatical cycles, in which case it cancels the generation of the parser.

6.13.2 Hidden left-recursions

When we have a non-terminal A such that:

$$A \stackrel{*}{\Rightarrow} \mu A \phi \tag{6.1}$$
$$\mu \stackrel{*}{\Rightarrow} \epsilon \tag{6.2}$$
$$\mu \neq \epsilon \tag{6.3}$$

`DURA` cannot generate a working parser either.

6.13.3 Size

There is no hard-coded limitation in terms of size for the grammar, the number of modules, etc. However, some of `DURA`'s algorithms do require vast amounts of dynamic memory. If `DURA` halts with an out of memory error message, one can increase the amount of dynamic memory by using the "`-ma`" command line option as supported by the plain YAFL compiler [31].

Figure 6.22 `dura.log`, the description of the push-down automaton

```
State: 0

Terminal transitions:
  From state 0 to 1 on lexeme Minus
  From state 0 to 2 on lexeme LParen
  From state 0 to 3 on lexeme Integer
  From state 0 to 4 on lexeme Ident
  From state 0 to 5 on lexeme Comma
Non-terminal transitions:
  From state 0 to 6 on nonterminal 2
  From state 0 to 7 on nonterminal 3
  From state 0 to 8 on nonterminal 4
  From state 0 to 9 on nonterminal 6
  From state 0 to 10 on nonterminal 7

State: 1
2:3:Minus *  Calc.UnaryExpr.TheArg!(2)

Terminal transitions:
  From state 1 to 1 on lexeme Minus
  From state 1 to 2 on lexeme LParen
  From state 1 to 3 on lexeme Integer
  From state 1 to 4 on lexeme Ident
  From state 1 to 5 on lexeme Comma
Non-terminal transitions:
  From state 1 to 26 on nonterminal 2
  From state 1 to 10 on nonterminal 7

State: 2
7:14:LParen *  THIS!(Calc.Expression:3) RParen

Terminal transitions:
  From state 2 to 1 on lexeme Minus
  From state 2 to 2 on lexeme LParen
  From state 2 to 3 on lexeme Integer
  From state 2 to 4 on lexeme Ident
  From state 2 to 5 on lexeme Comma
Non-terminal transitions:
  From state 2 to 6 on nonterminal 2
  From state 2 to 24 on nonterminal 3
  From state 2 to 8 on nonterminal 4
  From state 2 to 9 on nonterminal 6
  From state 2 to 10 on nonterminal 7

State: 3
7:15:Integer *
7:17: CalcLiteral.Literal.Im!Integer *

Terminal transitions:
Reductions:
  15/[43] Integer [44]<NOTNULL> Fetch non-terminal: Literal(0)
  17/[47]  CalcLiteral.Literal.Im!Integer [48]<NOTNULL> Fetch non-
terminal: Literal(0)
...
```

Chapter 7

`DURA` source code

This chapter contains the entire source code of DURA. It is a rather long, and undigested piece of technical literature. It would have been possible, in theory, to use the literate programming tools in such a way that the source code is presented in a way considered as more convenient to read, without necessarily basing it on the modular structure induced by YAFL. We did not take advantage of this opportunity, and kept the modular structure in place, producing one section for every module.

This source code includes three kinds of components:

- DURA itself, that is, the source code of the tool that will actually generate parsers
- Libraries, that is, modules that are meant to be linked with the generated parsers. This is mainly the case of the DuraBase module (See page 321).
- The skeleton (See page 865), which is not truly source code, but a text file that contains parts that are common to all parsers, but that could not be made reusable conveniently using inheritance. The skeleton includes specific codes that will be replaced by parser-specific items when the parser is to be generated.

7.1 Overall structure

Before getting to the gory details of the implementation, it is essential to understand that DURA is meant to be used in two phases:

- First, DURA is used to separately compile all modules of a project, including modules that contain grammatical information, as depicted in figure 7.1. DURA is an extension of the plain YAFL compiler, making extensive use of YAFL's compiler support, in such a way that grammar rules are recognized as part of class implementations. Hence, one can say that the first representation is a YAFL-compile time representation. Most if not error checking is performed at this level: referring to a non-existent rule, using a non-supported operator, etc... The grammatical information goes through several transformation stages, before being stored as an intermediate compilation with the extension ".cy".

- Then, DURA is used *again* to gather all the grammatical information that has been produced by the separate compilations, and produces a working parser, as depicted in figure 7.2.

DURA uses various intermediate representations before actually generating a working parser.

- DURA produces a compiler-independent representation out of this representation made of the extensions to the plain YAFL compiler (See module *DuraIntermediateFormat* on page 681). This compiler-independent representation reproduces all the intelligence of the abstraction level of the grammar writer, without transformation. Hence, this second form supports all the operators and constructs defined in the DURA user guide (See page 255). This intermediate representation is not connected to the YAFL compiler anymore. Basically, this representation could be generated by a tool that has no direct connection with the YAFL world.

- DURA supports the translation from this intermediate form to and from a lisp-like representation, that is merely used for permanent storage. Such lisp-like structures can be stored and read in a very convenient and more importantly, generic way (See module *DuraLisp* on page 796).

- Then, DURA also supports a more specific representation, referred to as the *flattened* representation (See module *DuraFlat* on page 474). This representation is the normalized form which is commonly used in most compiler textbooks. In this flattened representation, there are no more complex operators such as repetitions. All grammar rules are of the form $N ::= \mu$ where $\mu \in (N \cup T)*$. In this form, repetitions are replaced by explicit recursion. This last representation is attached to a specific parser generation scheme. It implements *items* (See module *DuraItem* on page 619), $FIRST$ and $FOLLOW$ sets, etc. Would one wish to generate LL(1) parsers instead of LR ones, the three first representations

Figure 7.1 Separate compilation of modules that contain grammatical information

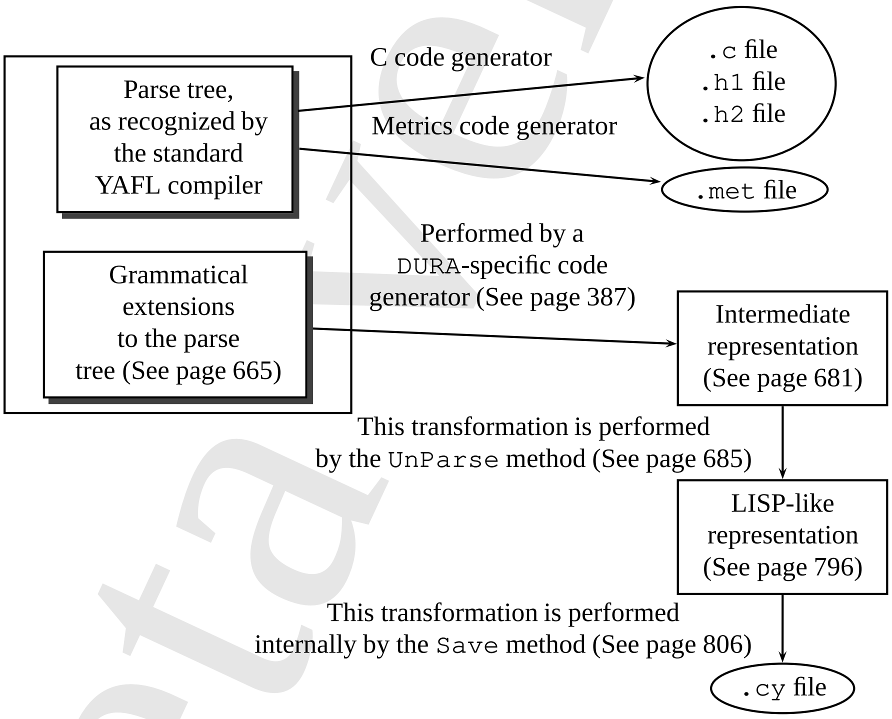
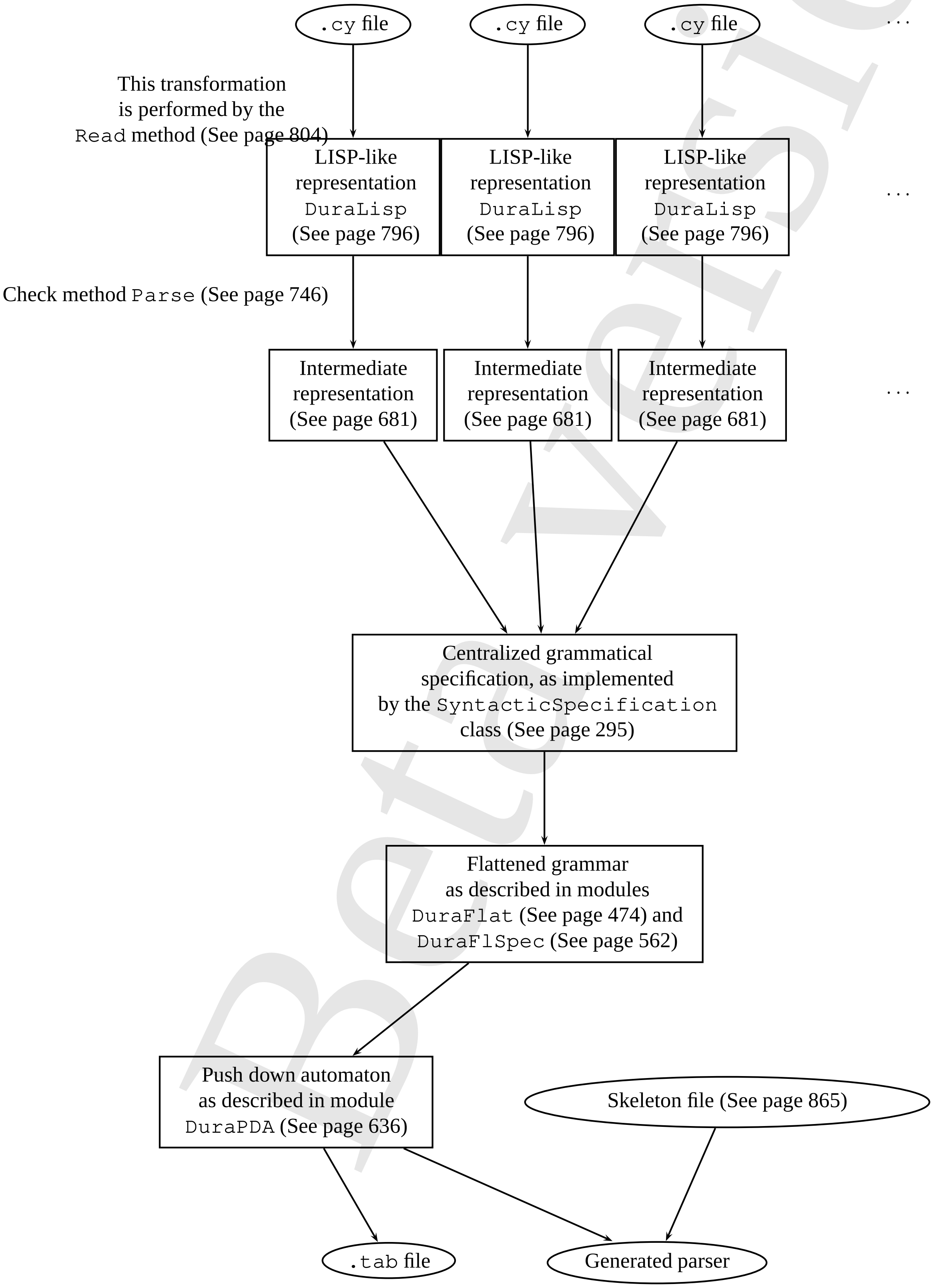


Figure 7.2 Producing a working parser



described above might be reused, while the fourth one, the last one before actual generation of a working parser, might have to be adapated or even rewritten from scratch.

The difference between the intermediate format (See module *DuraIntermediateFormat* on page 681) and the non-terminals designed as extensions of the plain YAFL compiler (See module *DuraGrClause* on page 665) is justified by mere implementation issues attached to the DURA model of a compiler-compiler. This model implies that the grammar definition can be *modular*. Hence, the number of modules that have to be read in order to gather a complete grammar is potentially unbounded, exceeding pragmatic memory limitations.

Therefore, instead of gathering all the grammar rules directly from their enclosing module, a two stages scheme is used. When a module is compiled, if one or more of its classes have grammar rules attached, an intermediate representation as described above is built, transformed into its equivalent lisp-like structure, and saved onto a specific file (See figure 7.1).

DURA's parsers are monolithic by nature, both for conveniency and performance reasons. Hence, generating them requires access to the entire grammar. However, using these intermediate representation, the grammar can be gathered efficiently without having to reload, parse and analyze all the source code of all the modules of the project.

Dividing the parser-generation process in several intermediate representations also serves other purposes than strictly operational ones. The different representations can be more or less suited to specific forms of processings. At the user-level, it is of course much easier and more convenient to deal with high-level operators and constructs, such as iteration. On the other hand, computing $FIRST$ or $FOLLOW$ sets directly on this high-level representation is very cumbersome, while it is straightforward when performed on the flattened representation.

7.2 Implementation issues

Most yacc (and lex, [56] being a notorious exception) implementations are table-driven, and take advantage of C's ability to initialize tables at compile time. YAFL does not offer this ability. For performance reasons, cool did not use external tables, but rather, produced the tables as intricate nested CASE statements.

This approach has been followed by DURA's early designs, but has soon shown pragmatic limitations. Compiling the generated parsers was becoming unpractical when the grammar grew beyond the size of toy languages.

In order to allow the convenient production of large parsers, DURA can now produce interpreted parsers as well (See page 256), where the automaton is produced in a compact form as an IntDictionary (See page 764) and saved with the "$.tab$" extension.

Given the flexibility of this external table approach, and the cost of maintaining a tool that can support both table-driven and compiled automata, it is likely that support for compiled automata will be discontinued in the near future.

7.3 YDura

`YDura` is the root module of `DURA` . It is not very long, nor apparently complex, but it does implement some of the most obscure constructions that have been used in the entire YAFL world.

Among other subtleties, it uses the externalized code generators as supported by YAFL to add a `DURA`-specific code generation pass to the compiler.

⟨*Definition* ⟩≡
```
DEFINITION MODULE YDura;

IMPORT Ycc;

  ⟨Dura root class definition 288a⟩

END YDura;
```

⟨*Implementation* ⟩≡
```
IMPLEMENTATION MODULE YDura;

FROM DuraCreators IMPORT DuraCodeGeneratorCreator,
                         DuraStreamCreator,
                         DuraObjectCreator,
                         DuraPragmaCreator;

FROM DuraSpecification IMPORT SyntacticSpecification;

IMPORT DuraLex;
FROM YaflCompiler IMPORT Compiler;
FROM YaflCreator IMPORT Creators, CodeGeneratorCreatorList;
FROM Streams IMPORT StdOut, StdErr;
FROM YaflCfg IMPORT YaflCfg;
IMPORT SYSTEM;
IMPORT DuraCfg;
IMPORT Profile;
FROM YaflProject IMPORT ProjectFile;

  ⟨Dura root class implementation 288b⟩

END YDura;
```

YAFL

Dura inherits from the common YAFL compiler Yc as provided by the Ycc module. It retains most of its base class structure and logic, and merely plugs additional processing by redefining a few methods.

288a ⟨Dura *root class definition* 288a⟩≡
```
CLASS Dura;
  INHERITS Ycc.Yc;

  CONST
    DuraVersion = ‘‘2.060”;

  REDEFINE METHOD CREATE;

END Dura;
```

288b ⟨Dura *root class implementation* 288b⟩≡
```
CLASS Dura;
  INHERITS Ycc.Yc;

  ⟨Adding processing to the inherited entry point 290a⟩

  ⟨Cleaning up memory 292b⟩

  ⟨Defining an ad hoc version disclaimer 288c⟩
  ⟨Defining a completed Help method 289a⟩
  ⟨Processing options 289b⟩

  ⟨Taking DURA code generation into account 292a⟩

  ⟨Compile a project, generate parser if required 293a⟩

END Dura;
```

The first improvement when compared to the original compiler is quite mundane. Disclaimer is redefined in such a way that DURA's version number is appended to YAFL's version number. Disclaimer is called by the compiler and displayed before anything else can be displayed.

288c ⟨*Defining an ad hoc version disclaimer* 288c⟩≡
```
REDEFINE METHOD Disclaimer: ARRAY OF CHAR;
  BEGIN
  RESULT ← ‘‘DURA Compiler/Compiler - Version ” +
               DuraVersion +
               ‘‘ (” + Ycc.Yc.Version + ‘‘)”;
  END Disclaimer;
```

The second improvement is not much more complicated. When displaying the list of options that can be passed to the compiler, `DURA`-specific options are listed as well.

289a ⟨*Defining a completed* `Help` *method* 289a⟩≡

```
REDEFINE METHOD Help;
  BEGIN
  BASE;
  l(“”);
  l(“                       DURA (Compiler generator) options”);
  l(“-lf<Fname>           :  Produce the interpreter’s table in <Fname>”);
  l(“-ll<Module>.<Class>:  Use class <Class> from module <Module> to ”);
  l(“                        gather the constants attached to the lexical”);
  l(“                        constants used in grammar rules”);
  l(“-lt<Module>.<Class>:  Generate the aggregated parser in”);
  l(“                        <Module>.<Class>”);
  l(“-lr<Module>.<Class>:  Use <Module>.<Class> as root non-terminal”);
  l(“                        for the generated parser”);
  l(“-lsd<FName>          :  Use <FName> as skeleton file for definition”);
  l(“                        modules”);
  l(“-lsi<FName>          :  Use <FName> as skeleton file for implementation”);
  l(“                        modules”);
  l(“-lv<num>             :  Set the generated parser’s verbosity level” +
                          “ to <num>”);
  l(“-lvd                 :  The generated parser’s verbosity level is” +
                          “ dynamic”);
  l(“-ln                  :  Disable the parser’s normalization”);
  l(“-lg                  :  Ignore all grammar info”);
  l(“-ly<FName>           :  Produce a yacc-equivalent parser in <fname>”);
  END Help;
```

When processing command-line options, `DURA`-specific options (`-lr`, `-lt`, `-ll`) are tested first by calling the `DuraCfg` once class (See page 370). If the option has not been recognized as being related to `DURA` , the original command-line option analyzer is used by calling `BASE`. YAFL

Please note that short-circuit lazy evaluation as provided by YAFL has been used here: if evaluating the first argument of the `OR` yields `TRUE`, the second is not evaluated at all.

289b ⟨*Processing options* 289b⟩≡

```
REDEFINE METHOD ProcessOption (Opt: ARRAY OF CHAR): BOOLEAN;
  BEGIN
  RESULT ← DuraCfg.ProcessOption(Opt) ∨ BASE(Opt);
  END ProcessOption;
```

When redefining the entry point for the compiler, several actions are taken before the original compiler is called by BASE.

290a ⟨*Adding processing to the inherited entry point* 290a⟩≡

```
REDEFINE METHOD CREATE;
   VAR
     PragCreator: DuraPragmaCreator;
   BEGIN
```

First, new keywords such as NONTERMINAL or GRAMMAR are defined, and added to the original lexical analyzer (See page 794).

290b ⟨*Adding processing to the inherited entry point* 290a⟩+≡

```
DuraLex.EnterNewKeywords;
```

Then, a specialized StreamCreator is sent to Creators so that whenever a new StreamLookAhead instance will be required, a DURA-specific one will be created and used. This specialized StreamLookAhead is needed to cope with the differences in terms of grammar between standard YAFL and DURA .

The DURA-aware creators are defined in the DuraCreators module (See page 404).

290c ⟨*Adding processing to the inherited entry point* 290a⟩+≡

```
Creators.SetStream (DuraStreamCreator);
```

Similarly, a specialized ObjectCreator is sent to Creators as well, so that specialized (understand: DURA-specialized) versions of YAFL parse tree instances are used instead of the original YAFL ones: using DuraClassDefinition instead of plain ClassDefinition, etc...

290d ⟨*Adding processing to the inherited entry point* 290a⟩+≡

```
Creators.SetObject (DuraObjectCreator);
IF Creators.Pragma ≠ Λ THEN
   ⟨Ensure the current PragmaCreator is of the proper class 291⟩
 ELSE
   PragCreator.CREATE;
   Creators.SetPragma (PragCreator);
   END;
BASE;
END CREATE;
```

As amazing as it might sound, at this stage, we've already started using DURA-derived compilers, that must be able to define their own extended PragmaCreators. For instance, RainCode (See page 920) is based on a specific compiler which is derived from DURA. Of course, for this scheme to work properly, DURA must avoid resetting its own creator without first checking whether a proper creator has been defined already.

If such as extended creator was defined, the following chunks makes sure that the creator is indeed an extension of the creator required by `DURA`, by first testing the class of the current pragma creator using a YAFL `WHAT` statement, and by setting a newly created creator if necessary.

291 ⟨*Ensure the current* `PragmaCreator` *is of the proper class* 291⟩≡

```
WHAT Creators.Pragma OF
  IN DuraPragmaCreator:
    END;
ELSE
  PragCreator.CREATE;
  Creators.SetPragma (PragCreator);
  END;
```

Code generation externalization is a strange feature of the YAFL compiler. It allows specific code generation passes to be added to the compiler without actually altering its source code in any way.

In practice, each node of the parse tree can be attached to a corresponding code generator which cannot be shared among several nodes of the parse tree. The parse tree is shadowed by a tree made of the corresponding code generators. The code generator's class depends on the node's class, that is, different code generators will be created for every different kind of node that will be used in the parse tree. Generating code for a node can then be summarized as:

- Checking whether the node is attached to a code generator.
- If no such code generator is attached to the node, create one and attach it.
- Use the code generator to actually generate code for the node.

Then, when the entire parse tree attached to an implementation module is compiled, all the code generators attached to all its nodes are removed, so that new, possibly different code generators can be created and used in an entirely different context.

This mechanism is automatic. The only thing that must be provided is a list of code generator creators. A code generator creator is an object that receives a node, and creates the corresponding code generator. For instance, when given a YAFL loop statement as parameter, the code generator creator attached to the `C` code generation will create code generator that can produce the sematically equivalent `C` loop.

By default, the standard YAFL compiler defines two code generators: the `C` code generator, and the metrics generator. An example of another kind of code generator could be a native code generator[1]. These code generators defines code generator creators that are gathered in a list that is elaborated by the `CodeGeneratorCreatorList` described below. Well, the only thing `DURA` has to do in order to provide its own code generation scheme is to add – not replace ! – its own code generator creator in this list.

[1] By the way, such a code generator has been developed [57] using the intermediate representation defined by the GNU C [81], C++ and Ada [82] compiler. This native code generation for YAFL uses the mechanism described here to attach external code generation schemes to the compiler.

The following chunk adds the DURA-specific code generator creator (See class *DuraCreators.DuraCodeGeneratorCreator* on page 406) to the list of code generator creators if global optimization is not enabled. Global optimization [23] aims at producing a space- and performance-optimized version of a system by considering a system as a whole rather than considering one module at a time. It is a lengthy process, which is generally used at the latest stage of development, just before delivery. At this stage, the grammar information as well as the resulting parser must have been generated already. Besides, the generation of grammar information for the entire project will not work if the system includes more than a one parser and/or more than one lexer, while there is no such restriction for common modular compilation.

292a ⟨*Taking* DURA *code generation into account* 292a⟩≡

```
REDEFINE METHOD BuildCodeGCList: CodeGeneratorCreatorList;
  BEGIN
  RESULT ← BASE;
  IF ¬ YaflCfg.PleaseGlobalOptimize THEN
    RESULT.Append (DuraCodeGeneratorCreator);
    END;
  END BuildCodeGCList;
```

When cleaning up memory, the references to definition modules that are found in the DuraCfg module must be cleared in order to let the garbage collector recollect whatever it can.

292b ⟨*Cleaning up memory* 292b⟩≡

```
REDEFINE METHOD CleanUpMemory;
  BEGIN
  BASE;
  DuraCfg.ResetClasses;
  END CleanUpMemory;
```

`ExecutePrj` redefines the method that compiles an entire project in the plain $\mathrm{Y^AF_L}$ compiler. This redefinition checks whether the user has requested a parser generation (by means of the `TargetModuleName` method (See page 386)) and proceeds normally, recompiling all modules that must be recompiled according to the dependancy relationships if no parser must be generated.

293a ⟨*Compile a project, generate parser if required* 293a⟩≡

```
REDEFINE METHOD ExecutePrj (Prj: ProjectFile): BOOLEAN;
    VAR
      Spec: SyntacticSpecification;
    BEGIN
    IF DuraCfg.TargetModuleName ≠ Λ THEN
      IF DuraCfg.RootNonTerminalClass ≠ Λ THEN
        ⟨Actually generate the parser for the project Prj 293b⟩
       ELSE
        StdErr.WriteLine (‘‘The root NONTERMINAL ” +
                          ‘‘must be specified”);
        END;
     ELSIF DuraCfg.RootNonTerminalClass ≠ Λ THEN
      StdErr.WriteLine (‘‘The target module name for the ” +
                        ‘‘parser must be specified”);
     ELSE
      ⟨No parser generation has been requested, just compile 293c⟩
      END;
    END ExecutePrj;
```

Generating the parser is merely a matter of creating an instance of the `SyntacticSpecification` class (See page 295), to hand it the project `Prj`, define the root non-terminal class (the base class for all non-terminal classes in the project), and to call `GenerateCode` (See page 306) that takes care of the rest.

293b ⟨*Actually generate the parser for the project* `Prj` 293b⟩≡

```
Spec.CREATE (Prj);
```

293c ⟨*No parser generation has been requested, just compile* 293c⟩≡

```
RESULT ← BASE (Prj);
```

7.4 DuraSpecification

This module implements the `SyntacticSpecification` class, which encapsulates the highest-level intelligence about parser generation in DURA. When `CREAT`ing a new instance of this class, a YAFL project must be given as parameter (See method *DuraSpecification.SyntacticSpecification.CREATE* on page 299) and that's about it. The rest happens automatically: the `SyntacticSpecification` will extract the grammatical information from all the classes defined within the project, produce the corresponding flattened representation (See method *DuraSpecification.SyntacticSpecification.FlatSpec* on page 302), then the automaton (See method *DuraSpecification.SyntacticSpecification.Automaton* on page 304), generate the parser files (See method *DuraSpecification.SyntacticSpecification.GenerateCode* on page 306), and the table files in case of an interpreted parser (See method *DuraSpecification.SyntacticSpecification.GenerateTables* on page 305), as depicted in figure 7.2 in page 285).

This is how it happens in terms of sequence, not in terms of method invocations. `FlatSpec` and `Automaton` are lazy methods, which compute resp. the flat specification and the push down automaton on first invocation, retaining the result in an attribute for efficient further retrieval:

- `CREATE` invokes `GenerateCode` directly, without bothering about things being initialized properly or not
- `GenerateCode` calls `Automaton` that has not been called before, so the automaton is built on the fly.
- While building this automaton, `FlatSpec` is invoked for the first time as well, so the flat representation is built on the fly as well.

⟨*Definition* ⟩≡
⟨`DuraSpecification` *definition import list* 319b⟩

⟨`SyntacticSpecification` *class definition* 295⟩

END `DuraSpecification;`

⟨*Implementation* ⟩≡
IMPLEMENTATION MODULE `DuraSpecification;`

⟨`DuraSpecification` *implementation import list* 320⟩

⟨`SyntacticSpecification` *class implementation* 297a⟩

END `DuraSpecification;`

295 ⟨`SyntacticSpecification` *class definition* 295⟩≡

```
CLASS SyntacticSpecification;
  REDEFINE METHOD CREATE (Prj: ProjectFile);
  METHOD NonTerminals: DeclList(DuraClassDefinition);
  METHOD Modules: NTList(DuraDefinitionModule);

  METHOD FindModule (ModName: ARRAY OF CHAR): DuraDefinitionModule;
  METHOD FindClass (ModName,
                    ClassName: ARRAY OF CHAR): DuraClassDefinition;
END SyntacticSpecification;
```

SyntacticSpecification

This class links the grammar, as a number of $FlatRule$s (See page 477), and the corresponding automaton (See page 651), by defining the correspondance rules between these two representations of a given reality.

$$Rules : \mathbb{P}\, FlatRule \text{ (See page 477)}$$
$$Root : \rho FlatRule$$

$$\delta Root \in Rules$$
$$Rules = \{x \in FlatRule \bullet \exists\, s \in \delta Root.Strings, r \in \rho FlatRule \bullet$$
$$r \in \operatorname{ran} s \land \delta r = x$$
$$\}$$

$$Automaton : PushDownAutomaton \text{ (See page 651)}$$
$$StartState, FinalState : \rho PDAState \text{ (See page 638)}$$

δ^1 (See page 979) is the generalization of the simple dereferencing operation δ to a set of references instead of an atomic reference.

$$Automaton.States = \{\delta StartState\} \cup \delta^1\, StartState.ReachableStates$$
$$\cup \{\delta FinalState\}$$

The start and final state of our automaton are defined in terms of the first and final items of the right hand sides attached to the root grammar rule.

$$\delta StartState.Items = \bigcup_{r \in \delta Root.RHS} \delta(r.FirstItem).Closure$$
$$\delta FinalState.Items = \bigcup_{r \in \delta Root.RHS} \delta(r.LastItem)$$

The $Goto$ function defined hereunder is a restriction of the *goto* function, as defined for LR automata in , in the sense that it does not operate on any $Symbol$ (See page 476), but on non-terminals only, as represented by the $FlatRule$ class (See page 477).

$$Goto : (\rho PDAState \times \rho FlatRule) \mapsto PDAState$$

$$\forall\, s \in \rho PDAState, r \in \rho FlatRule \bullet$$
$$Goto(s, r) = \delta s.Transition(r)$$

297a ⟨`SyntacticSpecification` *class implementation* 297a⟩≡

```
CLASS SyntacticSpecification;

    VAR
      TheNonTerminals: DeclList(DuraClassDefinition);
      TheModules: NTList(DuraDefinitionModule);
      ThePrj: ProjectFile;
      LogFile: OutputStream;

    ⟨Starting the generation of the parser by means of the CREATE method 299⟩

    METHOD NonTerminals: DeclList(DuraClassDefinition);
      BEGIN
      RESULT ← TheNonTerminals;
      END NonTerminals;

    ⟨Consulting the module list of the specification 297b⟩
    ⟨Finding modules and classes within a specification 298b⟩
    ⟨FlatSpec method implementation 302c⟩
    ⟨Automaton method implementation 304b⟩

    ⟨GenerateYacc method implementation 317a⟩
    ⟨GenerateCode method implementation 306b⟩
    ⟨GenerateTables method implementation 305a⟩
    ⟨GenerateImportList method implementation 306c⟩
    ⟨GenerateMetaInformationDeclaration method implementation 307a⟩
    ⟨GenerateMetaInformationInit method implementation 307b⟩
    ⟨GenerateCreationMethods method implementation 307c⟩

    ⟨Skeleton expansion method 309a⟩

  END SyntacticSpecification;
```

297b ⟨*Consulting the module list of the specification* 297b⟩≡

```
  METHOD Modules: NTList(DuraDefinitionModule);
    VAR
      Pos: INTEGER;
      Candidate: DuraDefinitionModule;
    BEGIN
    IF TheModules = Λ THEN
      ⟨Compute the module list, keep it in TheModules 298a⟩
      END;
    RESULT ← TheModules;
    END Modules;
```

This chunk does not compute the list of all modules, it computes the list of the modules which contain at least a non-terminal class. `TheNonTerminals` is initialized at the beginning of the `CREATE` method (See method *DuraSpecification.SyntacticSpecification.CREATE* on the facing page).

298a ⟨*Compute the module list, keep it in* `TheModules` 298a⟩≡

```
TheModules.CREATE;
FOR Nt ∈ TheNonTerminals DO
  Candidate ← Nt.Module;
  IF ∄ m ∈ TheModules • m = Candidate THEN
    TheModules.Insert (0, Candidate);
    END;
  END;
```

YAFL The following method makes use of YAFL quantifiers [83],and assumes that `ModName` has passed through the string normalization process that allows character strings to be compared by reference instead of value in the YAFL compiler [31].

298b ⟨*Finding modules and classes within a specification* 298b⟩≡

```
METHOD FindModule (ModName: ARRAY OF CHAR):
          DuraDefinitionModule;
  BEGIN
  RESULT ← FIRST Mod ∈ Modules • Mod.Id.Data = ModName;
  END FindModule;
```

Finding a class is not essentially different. It uses `GetClass` (See page 819) to find a class within a module.

298c ⟨*Finding modules and classes within a specification* 298b⟩+≡

```
METHOD FindClass (ModName,
                  ClassName: ARRAY OF CHAR):
                              DuraClassDefinition;
  VAR
    Mod: DuraDefinitionModule;
  BEGIN
  Mod ← FindModule (ModName);
  IF Mod ≠ Λ THEN
    RESULT ← Mod.GetClass (ClassName);
    END;
  END FindClass;
```

As noted in the preamble of this section, the `CREATE` method does it all. It first builds the initial data structures required to produce the parser, then calls `GenerateCode` (See page 306) which will build the intermediate representations on the fly.

299 ⟨*Starting the generation of the parser by means of the* `CREATE` *method* 299⟩≡

```
REDEFINE METHOD CREATE (Prj: ProjectFile);
  VAR
    Mod: DuraDefinitionModule;
    DefMod: DefinitionModule;
    PrjMod: ProjectModule;
    TargetRule: FlatRule;
    YaccOutput: OutputStream;
  BEGIN
  BASE;
  ThePrj ← Prj;
  TheNonTerminals.CREATE;
  ⟨Go through all the modules and classes, store whatever is needed 300a⟩
  ⟨Semantical analysis 301c⟩
  ⟨Open the log file 302a⟩
  Λ ← GenerateCode (LogFile);
  ⟨Generate tables if required 301d⟩
  ⟨Generate yacc file if required 301e⟩
  IF DuraCfg.VerboseLevel > 0 THEN
    Automaton.DumpStatistics;
    END;
  ⟨Close the log file 302b⟩
  END CREATE;
```

We first have to build our internal data structures, namely, the list of non-terminal classes we will use to produce a working parser. To do so, we visit all the modules in out system in sequence. For each module, we visit all the public classes it defines. YAFL

This loop visits the list backwards because we want to be able to remove an element from the list in the same pass without dealing with uncomfortable loop exit conditions.

300a ⟨*Go through all the modules and classes, store whatever is needed* 300a⟩≡

```
FOR i ∈ Prj.Modules.Size - 1 .. 0 BY -1 DO
  PrjMod ← Prj.Modules_i;
  IF PrjMod.TrueModule = Λ THEN
    ⟨Attach the proper YAFL module to PrjMod 300b⟩
    END;
  DefMod ← PrjMod.TrueModule;
  IF DefMod = Λ THEN
    ⟨Sadly, delete the module number i from the list 301a⟩
   ELSE
    WHAT DefMod OF
      IN DuraDefinitionModule:
        Mod ← TAG;
        END;
      END;
    ASSERT Mod ≠ Λ;
    ⟨Consider the classes defined in Mod, store nonterminal classes 301b⟩
    END;
  END;
```

YAFL

YAFL `ProjectFiles` do not maintain `DefinitionModules`, but proxy (and much smaller) objects that provide some version management facilities without having to read the entire definition module in memory.

The purpose of this chunk is to read the definition module, perform full semantic analysis *on the definition modules only* by calling `TagClosure` which tags the transitive closure of imported modules, and attach the definition module as the `TrueModule` of the corresponding proxy object.

300b ⟨*Attach the proper YAFL module to* `PrjMod` 300b⟩≡

```
DefMod ← ModuleTable.GetDefinitionModule
              (PrjMod.World.TrueWorld,
               PrjMod.Get);
IF DefMod ≠ Λ THEN
  DefMod.TagClosure;
  PrjMod.SetTrueModule (DefMod);
 ELSE
  WarningHandler.SetError (0, 0,
                           ‘‘Cannot open definition module ” +
                              PrjMod.Get);
  END;
```

In an ideal world, all the modules of our project should be well formed before we start generating parsers. However, the generated parser is also part of the project, and will not be well-formed until generated by DURA. To support this kind of transient state, we must be able to deal with not so complete project information when producing a parser.

301a ⟨*Sadly, delete the module number* `i` *from the list* 301a⟩≡

```
Prj.Modules.Delete(i);
```

`DuraClasses` (See page 819) only returns non-terminal classes, that is, classes that contain grammatical information to include in the generated parser.

301b ⟨*Consider the classes defined in* `Mod`*, store nonterminal classes* 301b⟩≡

```
FOR Cl ∈ Mod.DuraClasses DO
  ASSERT Cl.IsNonTerminal;
  TheNonTerminals.Append (Cl);
  END;
```

The responsibility of semantical analysis is deferred to the modules and classes, that will, in turn, refer to their attached intermediate representation.

`DuraTag` (See page 820) is used in order to distinguish this action from `Tag` which is the name for the semantical analysis method in the normal YAFL compiler.

301c ⟨*Semantical analysis* 301c⟩≡

```
FOR m ∈ Modules DO
  m.DuraTag (THIS);
  END;
```

If the proper flag has been set (See method *DuraCfg.DuraCfg.GenerateInterpretedParsers* on page 382), let's generate tables (See method *DuraSpecification.SyntacticSpecification.GenerateTables* on page 305).

301d ⟨*Generate tables if required* 301d⟩≡

```
IF DuraCfg.GenerateInterpretedParsers THEN
  GenerateTables;
  END;
```

If a target `yacc` filename has been specified (See method *DuraCfg.DuraCfg.YaccTargetFName* on page 385), let's produce an almost equivalent `yacc` source file (See method *DuraSpecification.SyntacticSpecification.GenerateYacc* on page 317).

301e ⟨*Generate* `yacc` *file if required* 301e⟩≡

```
IF DuraCfg.YaccTargetFName ≠ Λ THEN
  YaccOutput.CREATE;
  YaccOutput.Create (DuraCfg.YaccTargetFName,
                     YaccOutput.WriteAccess);
  GenerateYacc (YaccOutput);
  YaccOutput.Close;
  END;
```

We create "dura.log" which will contain grammatical information for debugging purposes as shown in figure 6.21 in page 277.

302a ⟨*Open the log file* 302a⟩≡
```
LogFile.CREATE;
LogFile.Create (‘‘dura.log”, LogFile.WriteAccess);
ASSERT LogFile.ErrorCode = LogFile.NoError;
```

302b ⟨*Close the log file* 302b⟩≡
```
LogFile.Close;
```

302c ⟨FlatSpec *method implementation* 302c⟩≡
```
VAR
  TheFlatSpec: FlatSpecification;

METHOD FlatSpec: FlatSpecification;

  METHOD Emit (Msg: ARRAY OF CHAR);
    BEGIN
    IF DuraCfg.VerboseLevel > 0 THEN
      StdOut.WriteLine (Msg);
      END;
    END Emit;

  BEGIN
  IF TheFlatSpec = Λ THEN
    ⟨First, let each class build its own information 302d⟩
    ⟨Then, gather it all together 303b⟩
    ⟨Perform various grammar manipulation 303c⟩
    ⟨Compute a couple of other things on the grammar 304a⟩
    END;
  RESULT ← TheFlatSpec;
  END FlatSpec;
```

We first call BuildIntrinsicFlatRule (See page 422) on each of the non-terminals defined in the grammar. Each non-terminal class now has a flattened grammar rule derived canonically from the grammar clause defined in the class.

302d ⟨*First, let each class build its own information* 302d⟩≡
```
Emit (‘‘Producing flattened representation”);
FOR Nt ∈ TheNonTerminals DO
  Nt.BuildIntrinsicFlatRule;
  END;
```

Then, we call `BuildInheritedFlatRule` (See page 424) so that each nonterminal's flattened grammar is augmented by the flattened grammars attached to its derived classes.

303a ⟨*First, let each class build its own information* 302d⟩+≡

```
Emit(“Applying reverse inheritance”);
FOR Nt ∈ TheNonTerminals DO
  Nt.BuildInheritedFlatRule;
  END;
```

Building a `FlatSpecification` with the entire grammar is simple: it is enough to provide it with the root grammar rule (See method *DuraFlSpec.FlatSpecification.SetRoot* on page 605), and to call `Close` (See page 606) which will follow the transitive closure of all references to other grammar for the flat specification to contain all the grammar rules that are reachable starting at the root grammar rule.

303b ⟨*Then, gather it all together* 303b⟩≡

```
TheFlatSpec.CREATE;
ASSERT DuraCfg.RootNonTerminalClass.Flat ≠ Λ;
TheFlatSpec.SetRoot(DuraCfg.RootNonTerminalClass.Flat);
TheFlatSpec.SetError (TRUE);
TheFlatSpec.Close;
```

We must now alter the flattened grammar, to take precedence and associativity into account (See method *DuraFlSpec.FlatSpecification.HandleAssociativity* on page 574) ...

303c ⟨*Perform various grammar manipulation* 303c⟩≡

```
Emit(“Handling associativity”);
TheFlatSpec.HandleAssociativity;
```

We're not home yet. We now reduce the size of the flattened grammar by merging equivalent grammar rules (See method *DuraFlSpec.FlatSpecification.MergeEquivalentRules* on page 608). When flattening the grammar rules, some synthetic rules are created in several equivalent occurences, and this process will replace the references to all these equivalent rules to references to a single rule of the equivalence class.

303d ⟨*Perform various grammar manipulation* 303c⟩+≡

```
Emit(“Merging equivalent rules”);
TheFlatSpec.MergeEquivalentRules;
```

We then compute nullability and reducibility for all grammar rules (See method *DuraFlSpec.FlatSpecification.ComputeRuleProperties* on page 598). Then, if everything went without a hitch, we compute the various sets of terminals attached to each grammar rule (See method *DuraFlSpec.FlatSpecification.ComputeAttachedSets* on page 602). Finally, we compute the production set for each grammar rule, checking for grammatical cycles in the process (See method *DuraFlSpec.FlatSpecification.ComputeProductionSets* on page 601).

304a ⟨*Compute a couple of other things on the grammar* 304a⟩≡
```
Emit (‘‘Computing rule properties”);
IF TheFlatSpec.ComputeRuleProperties THEN
  Emit (‘‘Computing FIRST, LAST, PRECEDE, MIDDLE and FOLLOW sets”);
  TheFlatSpec.ComputeAttachedSets;
  Emit (‘‘Computing production sets”);
  IF TheFlatSpec.ComputeProductionSets THEN
    Emit (‘‘Done”);
    TheFlatSpec.SetError (FALSE);
    END;
  END;
```

`Automaton` calls `ComputeAutomaton` (See page 574) to actually build the push down automaton, and dumps various pieces of information onto the log file “`dura.log`”.

304b ⟨`Automaton` *method implementation* 304b⟩≡
```
VAR
  TheAutomaton: PushDownAutomaton;

METHOD Automaton: PushDownAutomaton;
  BEGIN
  IF TheAutomaton = Λ THEN
    TheAutomaton ← FlatSpec.ComputeAutomaton (LogFile);
    IF DuraCfg.VerboseLevel > 0 THEN
      StdOut.WriteLine (‘‘Dumping the flat grammar”);
      END;
    FlatSpec.Dump (LogFile);
    LogFile.WriteLn;
    IF DuraCfg.VerboseLevel > 0 THEN
      StdOut.WriteLine (‘‘Dumping the automaton”);
      END;
    TheAutomaton.Dump (LogFile);
    IF DuraCfg.VerboseLevel > 0 THEN
      StdOut.WriteLine (‘‘Dumping the automaton, done”);
      END;
    END;
  RESULT ← TheAutomaton;
  END Automaton;
```

`GenerateTables` saves the parser's information in tabular form, in a text file which name was given on the command-line. It uses an `IntDictionary` (See page 763) to store complex arrays with a potentially high level of redundancy.

There is dirty detail here one must keep in mind. The automaton must have been generated and stored into the `IntDictionary` before the GOTO table can be stored, because the GOTO table relies on information that is stored in the internal structures describing rules when computing the automaton. Better to say it than to forget it...

305a ⟨`GenerateTables` *method implementation* 305a⟩≡

```
METHOD GenerateTables;
  VAR
    Dict: IntDictionary;
  BEGIN
  IF DuraCfg.VerboseLevel > 0 THEN
    StdOut.WriteLine (“Generating tables”);
    END;
  Dict.CREATE;
  ⟨Save data and names for the various tables 305b⟩
  Dict.Save (DuraCfg.ParseTableFileName);
  IF DuraCfg.VerboseLevel > 0 THEN
    StdOut.WriteLine (“Generating tables, done”);
    END;
  END GenerateTables;
```

The tables that describe the push down automaton are not written directly onto a file. They are gathered in an `IntDictionary` (See page 764) which manages them efficiently by sharing common arrays, and which can write them onto disk files for later retrieval.

`IntDictionary` also provides a `SaveName` method (See page 785) which associates an integer value with a character string, to allow the retrieval of tables in the `IntDictionary` using keys.

`StoreRecoveryInfo` (See page 594) takes care of the storage of its names as well.

305b ⟨*Save data and names for the various tables* 305b⟩≡

```
FlatSpec.StoreRecoveryInfo (Dict, Automaton);
```

`StoreArray` (See page 656) stores the transition table information into `Dict` and returns an index that can be used afterwards to retrieve it.

305c ⟨*Save data and names for the various tables* 305b⟩+≡

```
Dict.SaveName (“Automaton”, Automaton.StoreArray(Dict));
```

The following chunk stores the arrays returned by `RHSToRuleNrTable` (See page 598), `LengthTable` (See page 598) and `NullableTable` (See page 598).

306a ⟨*Save data and names for the various tables* 305b⟩+≡

```
Dict.SaveName (‘‘RHSToRuleNr”,
                    Dict.StoreArray(FlatSpec.RHSToRuleNrTable));
Dict.SaveName (‘‘Length”,
                    Dict.StoreArray(FlatSpec.LengthTable));
Dict.SaveName (‘‘Nullable”,
                    Dict.StoreArray(FlatSpec.NullableTable));

Dict.SaveName (‘‘Goto”,
                 FlatSpec.StoreGotoTable(Dict, Automaton.Size));
```

`GenerateCode` elaborates the target file names for the definition as well as for the implementation module, and expands the corresponding skeletons by calling `ExpandSkeleton` (See page 309).

306b ⟨`GenerateCode` *method implementation* 306b⟩≡

```
METHOD GenerateCode (LogFile: OutputStream): BOOLEAN;
  VAR
    MName,
    DefFName, ModFName: ARRAY OF CHAR;
  BEGIN
  MName ← DuraCfg.TargetModuleName;
  DefFName ← ThePrj.RootModule.World.
             TrueWorld.BuildSourceFName (MName,
                                         YaflCfg.DefExt);
  ModFName ← ThePrj.RootModule.World.
             TrueWorld.BuildSourceFName (MName,
                                         YaflCfg.ImpExt);
  RESULT ← ExpandSkeleton (DuraCfg.ImplementationSkeletonName,
                           ModFName)
                               ∧
           ExpandSkeleton (DuraCfg.DefinitionSkeletonName,
                           DefFName);
  END GenerateCode;
```

`GenerateImportList` is used when expanding the skeletons to produce a valid YAFL import list which imports all non-terminal classes defined in the project file.

306c ⟨`GenerateImportList` *method implementation* 306c⟩≡

```
METHOD GenerateImportList (Output: DuraOutputStream);
  BEGIN
  FOR m ∈ Modules DO
    Output.WriteLine(‘‘IMPORT ” + m.Id.Data + ‘‘;”);
    END;
  END GenerateImportList;
```

`GenerateMetaInformationDeclaration` is used when expanding the skeletons to produce a number of valid YAFL attribute declarations, which will refer to the data dictionary in order to be able to access attributes bypassing encapsulation. The actual work is performed by `GenerateMetaClassVariableDecl` (See page 428).

307a ⟨`GenerateMetaInformationDeclaration` *method implementation* 307a⟩≡

```
METHOD GenerateMetaInformationDeclaration (Output: DuraOutputStream);
  BEGIN
  FOR x ∈ TheNonTerminals DO
    x.GenerateMetaClassVariableDecl (Output);
    END;
  END GenerateMetaInformationDeclaration;
```

`GenerateMetaInformationInit` is used when expanding the skeletons to initialize the attributes declared by `GenerateMetaInformationDeclaration` above. The actual work is performed by `GenerateMetaClassVariableInit` (See page 430).

307b ⟨`GenerateMetaInformationInit` *method implementation* 307b⟩≡

```
METHOD GenerateMetaInformationInit (Output: DuraOutputStream);
  BEGIN
  FOR x ∈ TheNonTerminals DO
    x.GenerateMetaClassVariableInit (Output);
    END;
  END GenerateMetaInformationInit;
```

`GenerateCreationMethods` generate the body of methods that `CREATE` new instances of the various classes the `DURA`-generated parser must be able to allocate. First, the parser must be able to create instances of all the non-terminal classes defined in the project. (See method *DuraClDefinition.DuraClassDefinition.GenerateCreationMethodBody* on page 431).

307c ⟨`GenerateCreationMethods` *method implementation* 307c⟩≡

```
METHOD GenerateCreationMethods (Output: DuraOutputStream);
  VAR
    GeneratedLists: ArrayList(InterListType);
    TheClause: InterClause;
  BEGIN
  GeneratedLists.CREATE;
  FOR x ∈ TheNonTerminals DO
    x.GenerateCreationMethodBody (Output);
    ⟨Consider all list types, generate if appropriate 308⟩
    END;
  END GenerateCreationMethods;
```

Then, all sorts of lists (inheriting from `NonTerminalList` (See page 360)) can also be created by a DURA-generated parser, and creation methods must be created for them as well (See method *DuraIntermediateFormat.InterListType.GenerateCreationMethodBody* on page 700). To gather all these list types, all the intermediate grammar clauses attached to non-terminal classes are visited in sequence (See method *DuraClDefinition.DuraClassDefinition.InterGrammarClause* on page 418), and checked for the list types they refer to (See method *DuraIntermediateFormat.InterClause.ListTypes* on page 752).

308 ⟨*Consider all list types, generate if appropriate* 308⟩≡

```
TheClause ← x.InterGrammarClause;
IF TheClause ≠ Λ THEN
  FOR Candidate ∈ TheClause.ListTypes |
    ∄x ∈ GeneratedLists • x.IsEqual(Candidate) DO
    Candidate.GenerateCreationMethodBody (Output);
    GeneratedLists.Append (Candidate);
    END;
  END;
```

ExpandSkeleton takes a skeleton file (See page 865), and copies it to produce a working parser, replacing patterns by grammar-specific pieces of intelligence.

309a ⟨*Skeleton expansion method* 309a⟩≡

```
METHOD ExpandSkeleton (SourceFName, TargetFName: ARRAY OF CHAR): BOOLEAN;
  VAR
    Output: DuraOutputStream;
    TheLex: SkelLex;
    LineNr: INTEGER;

    ⟨CheckCondition utility method 310a⟩
    ⟨ExpandSingleToken utility method 311a⟩
    ⟨Error utility method 310b⟩

  BEGIN
  TheLex.CREATE;
  IF TheLex.AttachFile (SourceFName) ∨
     TheLex.AttachFile (SYSTEM.GetEnvironment (''YAFL_HOME'') +
                SYSTEM.DirSeparator + SourceFName) THEN
    ⟨Open Output 309b⟩
    WHILE TheLex.CurrentToken ≠ TheLex.EOF DO
      TheLex.Advance;
      ExpandSingleToken (TheLex, Output);
      END;
    Output.Close;
    RESULT ← TRUE;
   ELSE
    StdErr.WriteLine (''Cannot open file '' + SourceFName +
                  '' for input'');
    END;
  END ExpandSkeleton;
```

YAFL

Output is declared as a DuraOutputStream (See page 859) which inherits from VersionedOutputStream, provided by the YAFL standard library. AttemptCreate takes two file names as parameters. It will use the second file name as a temporary file for writing, and upon closing, it will check whether this second file is different from the first one. If not, the temporary file is discarded. Otherwise, it replaces the original file. This scheme has been set up to prevent time stamps attached to files to be altered while their content remains unchanged, thereby requiring large and useless recompilations.

309b ⟨*Open* Output 309b⟩≡

```
Output.CREATE;
Output.AttemptCreate (TargetFName, TargetFName + ''$'');
```

`CheckCondition` is called after that a `$If` command has been found in the skeleton, and tests for the various conditions listed in table 7.5 in page 865.

310a ⟨`CheckCondition` *utility method* 310a⟩≡

```
METHOD CheckCondition (TheLex: SkelLex): BOOLEAN;
  BEGIN
  WHILE TheLex.CurrentToken = TheLex.WhiteSpace DO
    TheLex.Advance;
    END;
  CASE TheLex.CurrentToken OF
    TheLex.TableDriven:
      RESULT ← DuraCfg.GenerateInterpretedParsers;
      END;
    TheLex.CaseDriven:
      RESULT ← ¬ DuraCfg.GenerateInterpretedParsers;
      END;
    TheLex.Debug:
      RESULT ← FALSE;
      END;
    TheLex.DynamicVerboseLevel:
      RESULT ← DuraCfg.GenerateDynamicVerbose;
      END;
   ELSE
    Error (‘‘Unrecognized condition: ” + TheLex.Image);
    TheLex.Advance;
    END;
  END CheckCondition;
```

310b ⟨`Error` *utility method* 310b⟩≡

```
VAR
  ErrorFound: BOOLEAN;

METHOD Error (Msg: ARRAY OF CHAR);
  BEGIN
  IF ¬ ErrorFound THEN
    ErrorFound ← TRUE;
    StdErr.WriteString (Msg + ‘‘ in line ”);
    StdErr.WriteInt (LineNr, 0);
    StdErr.WriteLn;
    END;
  END Error;
```

`ExpandSingleToken` does exactly what its name suggests: it expands a single token extracted from the skeleton file. The lexer `TheLex` it received as parameter is created by `lexyt` using the lexical specification in page 318. Quite logically, the various codes `ExpandSingleToken` recognizes are listed in the very same lexical specification.

311a ⟨`ExpandSingleToken` *utility method* 311a⟩≡

```
METHOD ExpandSingleToken (TheLex: SkelLex;
                          Output:  DuraOutputStream);
  VAR
    Cond, GoOn: BOOLEAN;
    Level: INTEGER;
  BEGIN
  CASE TheLex.CurrentToken OF
    ⟨First, consider control management tokens 311b⟩
    ⟨Depending on the input token, take appropriate action 313b⟩
   ELSE
    IF TheLex.Image ≠ Λ THEN
      Output.WriteString (TheLex.Image);
      END;
    END;
  END ExpandSingleToken;
```

The `$If` is the only control structure we support when expanding skeletons. `$Else` and `$EndIf` cannot be found here, they can only occur as part of a `$If` structure.

311b ⟨*First, consider control management tokens* 311b⟩≡

```
TheLex.If:
  ⟨Handle the $If in skeleton files 312a⟩
  END;
TheLex.Else:
  Error (“Unexpected token: $Else”);
  END;
TheLex.EndIf:
  Error (“Unexpected token: $EndIf”);
  END;
```

We basically check whether the condition that follows the `$If` (See method *DuraSpecification.SyntacticSpecification.CheckCondition* on page 310), and go through the skeleton file, `Cond` indicates whether what we are currently analyzing must be copied onto the target file or not, and we call `ExpandSingleToken` (See the preceding page) recursively for the actual expansion of the tokens.

312a ⟨*Handle the* `$If` *in skeleton files* 312a⟩≡

```
TheLex.Advance;
Cond ← CheckCondition (TheLex);
TheLex.Advance;
GoOn ← TRUE;
Level ← 1;
WHILE GoOn DO
  CASE TheLex.CurrentToken OF
    ⟨Check the various control tokens that can occur here 312b⟩
   ELSE
    IF Cond THEN
      ExpandSingleToken (TheLex, Output);
     ELSIF TheLex.CurrentToken = TheLex.NewLine THEN
      LineNr ← LineNr + 1;
      END;
    END;
  TheLex.Advance;
  GoOn ← GoOn ∧ (Level > 0);
  END;
```

This chunks handles nesting of `$If` structures, `$Else`, and the detection of `$EndIf`.

312b ⟨*Check the various control tokens that can occur here* 312b⟩≡

```
TheLex.Else:
  IF Level = 1 THEN
    Cond ← ¬ Cond;
    END;
  END;
TheLex.EndIf:
  Level ← Level - 1;
  END;
TheLex.EOF:
  Error (“Unexpected EOF”);
  GoOn ← FALSE;
  END;
```

If we are not currently skipping the input (understand: if `Cond` is `TRUE`), we call `ExpandSingleToken` recursively (See page 311) which will test for the condition specified after the `If` keyword, etc.

313a ⟨*Check the various control tokens that can occur here* 312b⟩+≡

```
TheLex.If:
  IF Cond THEN
    ExpandSingleToken (TheLex, Output);
   ELSE
    Level ← Level + 1;
    END;
  END;
```

Here follows the list of the patterns that can be found in the skeleton file, and the corresponding actions that must be taken when encountering them.

313b ⟨*Depending on the input token, take appropriate action* 313b⟩≡

```
TheLex.RootClass:
  Output.WriteString (DuraCfg.RootNonTerminalClass.Id.Data);
  END;
TheLex.RootModule:
  Output.WriteString (DuraCfg.RootNonTerminalClass.
                      Module.Id.Data);
  END;
TheLex.LexerClass:
  Output.WriteString (DuraCfg.LexicalAnalyzerClass.
                      Id.Data);
  END;
TheLex.LexerModule:
  Output.WriteString (DuraCfg.LexicalAnalyzerClass.
                      Module.Id.Data);
  END;
TheLex.ParserClass:
  Output.WriteString (DuraCfg.TargetModuleName);
  END;
TheLex.ParserModule:
  Output.WriteString (DuraCfg.TargetClassName);
  END;
```

The patterns that follow are not replaced by simple character strings, but by more complex structures generated by ancillary methods.

(See method *DuraSpecification.SyntacticSpecification.GenerateImportList* on page 306)

313c ⟨*Depending on the input token, take appropriate action* 313b⟩+≡

```
TheLex.ImportList:
  GenerateImportList (Output);
  END;
```

(See method *DuraFlSpec.FlatSpecification.GenerateFetchAttributes* on page 587)

314a ⟨*Depending on the input token, take appropriate action* 313b⟩+≡

```
  TheLex.FetchAttributes:
    FlatSpec.GenerateFetchAttributes (Output);
    END;
```

(See method *DuraFlSpec.FlatSpecification.GenerateReduceMethods* on page 582)

314b ⟨*Depending on the input token, take appropriate action* 313b⟩+≡

```
  TheLex.ReduceMethods:
    FlatSpec.GenerateReduceMethods (Output);
    END;
```

(See method *DuraFlSpec.FlatSpecification.GenerateGotoMethods* on page 582)

314c ⟨*Depending on the input token, take appropriate action* 313b⟩+≡

```
  TheLex.GotoMethods:
    FlatSpec.GenerateGotoMethods (Output);
    END;
```

(See method *DuraPDA.PushDownAutomaton.GenerateStateMachine* on page 654)

314d ⟨*Depending on the input token, take appropriate action* 313b⟩+≡

```
  TheLex.StateMachine:
    Automaton.GenerateStateMachine (Output);
    END;
```

(See method *DuraFlSpec.FlatSpecification.GenerateRHSToRule* on page 585)

314e ⟨*Depending on the input token, take appropriate action* 313b⟩+≡

```
  TheLex.RHSToRule:
    FlatSpec.GenerateRHSToRule (Output);
    END;
```

(See method *DuraSpecification.SyntacticSpecification.GenerateMetaInformation-Declaration* on page 307)

314f ⟨*Depending on the input token, take appropriate action* 313b⟩+≡

```
  TheLex.MetaInformationDeclaration:
    GenerateMetaInformationDeclaration (Output);
    END;
```

(See method *DuraSpecification.SyntacticSpecification.GenerateMetaInformation-Init* on page 307)

314g ⟨*Depending on the input token, take appropriate action* 313b⟩+≡

```
  TheLex.MetaInformationInit:
    GenerateMetaInformationInit (Output);
    END;
```

(See method *DuraSpecification.SyntacticSpecification.GenerateCreationMethods* on page 307)

314h ⟨*Depending on the input token, take appropriate action* 313b⟩+≡

```
  TheLex.CreationMethods:
    GenerateCreationMethods (Output);
    END;
```

(See method *DuraFlSpec.FlatSpecification.GenerateGenericGotoMethod* on page 591)

315a ⟨*Depending on the input token, take appropriate action* 313b⟩+≡

```
TheLex.GenericGotoMethod:
  FlatSpec.GenerateGenericGotoMethod (Output);
  END;
```

(See method *DuraFlSpec.FlatSpecification.GenerateIsVoidMethod* on page 592)

315b ⟨*Depending on the input token, take appropriate action* 313b⟩+≡

```
TheLex.IsVoidMethod:
  FlatSpec.GenerateIsVoidMethod(Output);
  END;
```

(See method *DuraFlSpec.FlatSpecification.GenerateGenericReduceMethod* on page 591)

315c ⟨*Depending on the input token, take appropriate action* 313b⟩+≡

```
TheLex.GenericReduceMethod:
  FlatSpec.GenerateGenericReduceMethod (Output);
  END;
```

Continued, again, again...

316 ⟨*Depending on the input token, take appropriate action* 313b⟩+≡

```
TheLex.MaxStates:
  Output.WriteInt (Automaton.Size, 0);
  END;
TheLex.MaxRules:
  Output.WriteInt (FlatSpec.Size, 0);
  END;
TheLex.MaxRHS:
  Output.WriteInt (FlatSpec.LastRHSNr, 0);
  END;
TheLex.NewLine:
  LineNr ← LineNr + 1;
  Output.WriteLn;
  END;
TheLex.Version:
  Output.WriteString (“’” + Dura.DuraVersion + “’”);
  END;
TheLex.TableFileName:
  Output.WriteString (“’” + DuraCfg.ParseTableFileName + “’”);
  END;
TheLex.VerboseLevel:
  IF DuraCfg.GenerateDynamicVerbose THEN
    Output.WriteString (“VerboseLevel”);
   ELSE
    Output.WriteInt (DuraCfg.GenerateVerboseLevel, 0);
    END;
  END;
TheLex.OtherKeyword:
  Error (“Unrecognized keyword ” + TheLex.Image);
  END;
```

`GenerateYacc` generates a `yacc` equivalent grammar, generating `%token` declarations for all the terminals defined in the lexer, and calling `GenerateYacc` (See page 509) to produce the `yacc` equivalents for all the rules defined in the flat representation.

317a ⟨`GenerateYacc` *method implementation* 317a⟩≡

```
METHOD GenerateYacc (Output: OutputStream);

  ⟨GenerateTokenDefs method implementation 317b⟩

  BEGIN
  GenerateTokenDefs (Output);
  Output.WriteLn;
  Output.WriteLine (“%%”);
  Output.WriteLn;
  FOR Rule ∈ FlatSpec DO
    Rule.GenerateYacc (Output);
    END;
  END GenerateYacc;
```

`GenerateTokenDefs` generate the header of the equivalent `yacc` parser, where the various `%token`s are defined. It uses `LexemeImage` (See page 379) to convert the lexeme's code into a more understandable identifier.

317b ⟨`GenerateTokenDefs` *method implementation* 317b⟩≡

```
METHOD GenerateTokenDefs (Output: OutputStream);
  VAR
    Count: INTEGER;
    a: ARRAY OF CHAR;
  CONST
    PerLine = 5;
  BEGIN
  FOR i ∈ DuraCfg.FirstToken .. DuraCfg.LastToken DO
    a ← String.Strip(DuraCfg.LexemeImage (i));
    IF Count MOD PerLine = 0 THEN
      Output.WriteString (“%token ”);
      END;
    Count ← Count + 1;
    Output.WriteString (a + “ ”);
    IF Count MOD PerLine = 0 THEN
      Output.WriteLn;
      END;
    END;
  Output.WriteLn;
  Output.WriteLine (“%token EOF”);
  Output.WriteLn;
  Output.WriteLn;
  END GenerateTokenDefs;
```

The lexical specification used to produce the lexer to expand the skeletons. It is not valid YAFL code. It is a `lexyt` lexical specification, which is used to produce an actual working lexer used in `ExpandSleleton` (See page 309).

318 ⟨"duraskel.yl" 318⟩≡

```
$BACKTRACKLEVEL 0

FetchAttributes    $FetchAttributes
ImportList                 $ImportList
StateMachine               $StateMachine
RootClass                  $RootClass
RootModule                 $RootModule
RHSToRule                  $RHSToRule
ParserClass                $ParserClass
ParserModule               $ParserModule
LexerClass                 $LexerClass
LexerModule        $LexerModule
CreationMethods    $CreationMethods
ReduceMethods              $ReduceMethods
GotoMethods                $GotoMethods
IsVoidMethod               $IsVoidMethod
AttributeInitialization $AttributeInitialization
GenericReduceMethod     $GenericReduceMethod
GenericGotoMethod       $GenericGotoMethod
MaxStates                  $MaxStates
MaxRHS                     $MaxRHS
MaxRules                   $MaxRules
MetaInformationDeclaration $MetaInformationDeclaration
MetaInformationInit     $MetaInformationInit
Version            $Version
VerboseLevel               $VerboseLevel
```

319a ⟨"duraskel.yl" 318⟩+≡
```
If                          $If
Else                        $Else
EndIf                       ($EndIf)|($Endif)

TableDriven                 $TableDriven
CaseDriven                  $CaseDriven
Debug                       $Debug
DynamicVerboseLevel      $DynamicVerbose(Level|@)

TableFileName               $TableFileName

Ident                    [_a-zA-Z][a-zA-Z0-9_]*
OtherKeyword                ${Ident}

Comment             \-\-.*
WhiteSpace          [ \t]
Other                        .
NewLine                     \n
```

319b ⟨DuraSpecification *definition import list* 319b⟩≡
```
DEFINITION MODULE DuraSpecification;

FROM Streams IMPORT OutputStream;

FROM YaflNTList IMPORT DeclList, NTSet, NTList;
FROM YaflProject IMPORT ProjectFile;

FROM DuraClDefinition IMPORT DuraClassDefinition;
FROM DuraOutput IMPORT DuraOutputStream;
FROM DuraModules IMPORT DuraDefinitionModule;
```

320 ⟨DuraSpecification *implementation import list* 320⟩≡

```
IMPORT ArrayList;
IMPORT DuraCfg;
FROM DuraFlat IMPORT FlatRule;
FROM DuraFlSpec IMPORT FlatSpecification;
FROM DuraIntDictionary IMPORT IntDictionary;
FROM DuraIntermediateFormat IMPORT InterListType, InterClause;
FROM DuraPDA IMPORT PushDownAutomaton;
FROM YDura IMPORT Dura;

IMPORT SkelLex;

IMPORT SYSTEM;
IMPORT String;
IMPORT Stack;
FROM Streams IMPORT StdOut, StdErr;

IMPORT ModuleTable;
FROM YaflCfg IMPORT YaflCfg;
FROM YaflError IMPORT WarningHandler;
FROM YaflModules IMPORT DefinitionModule;
FROM YaflProject IMPORT ProjectModule;
```

7.5 DuraBase

This module includes the definition of various bases classes which are derived in order to produce working parsers.

These classes are:

- `BaseParser` (See page 323) is the base class for all `DURA`-generated parsers.
- `ParserStackElement` (See page 332) is the base class for the elements of the LALR stack which is used by `DURA`-generated parsers.
- `ParserStackElement`-derived classes (See classes *DuraBase.ShiftStackElement*on page 335 and *DuraBase.ReduceStackElement*on page 337), that support different forms of behaviour for the LALR stack.
- `BaseNonTerminal` (See page 346) is the base class for all non-terminals. `DURA` makes sure that all classes defined as being `NONTERMINAL` classes derive directly or indirectly from `BaseNonTerminal`. This class provides several generic services, such as textual identification, subtree management, etc...
- `NonTerminalList` (See page 360) is the base class for all *collections* of non-terminals. For instance, when declaring that a function call takes 0 or more actual parameters, these actuals must be stored in a `NonTerminalList` if one wishes that `DURA` handles the aggregation of the non-terminals automatically. As a matter of fact, any `NonTerminalList`-derived classes can be used. For instance, one can declare a class `StatementList` that inherits directly or indirectly from `NonTerminalList`, declare attributes as being of this class, and fetch sequences of statements onto these attributes.
- `Walker` (See page 365) is an iterator class that can walk through an entire of a part of a parse tree, derecursifying the walkthrough by maintaining an internal stack.
- `ParserStatAcc` (See page 361) that accumulates statistical information about a parsing process, including which rules have been applied, the number of lines, etc.

⟨*Definition* ⟩≡

```
DEFINITION MODULE DuraBase;

IMPORT ArrayList;
FROM YLexicalAnalyzer IMPORT YLexicalAnalyzer,
                             AbstractCharStream;
FROM Streams IMPORT OutputStream;
FROM MetaClass IMPORT MetaClass, Attribute;
FROM DuraIntDictionary  IMPORT IntDictionary;

  ⟨BaseParser class definition 323a⟩
  ⟨BaseNonTerminal class definition 346c⟩
```

```
    ⟨NonTerminalList class definition 360b⟩

    ⟨Elements to store on the evaluation stack, definition 331⟩

    ⟨ParserStatAcc class definition 361b⟩
    ⟨Walker class definition 365b⟩
  END DuraBase;
```

⟨*Implementation* ⟩≡

```
  IMPLEMENTATION MODULE DuraBase;

  FROM Conversions IMPORT IntConversions;
  FROM List IMPORT List;
  FROM Streams IMPORT StdErr, StdOut;
  FROM MetaClass IMPORT Attribute, IntegerAttribute, BooleanAttribute,
                        ClassAttribute, StringAttribute;
  IMPORT SYSTEM;
  FROM YLexicalAnalyzer IMPORT AbstractCharStream;
  IMPORT Stack;

    ⟨BaseParser class implementation 324c⟩
    ⟨BaseNonTerminal class implementation 347a⟩
    ⟨NonTerminalList class implementation 361a⟩

    ⟨Elements to store on the evaluation stack, implementation 332a⟩

    ⟨ParserStatAcc class implementation 362a⟩
    ⟨Walker class implementation 366⟩

  END DuraBase;
```

Defining the `BaseParser` class as a parameterized class is a formal nicety, without much practical interest, beside the ability to define the `Parse` method in a generic way.

323a ⟨`BaseParser` *class definition* 323a⟩≡
```
CLASS BaseParser (RootClass IN BaseNonTerminal,
                  LexerClass IN YLexicalAnalyzer);


  METHOD DumpNonTerminal (Output: OutputStream;
                          Nt: BaseNonTerminal;
                             Lex: YLexicalAnalyzer;
                          Full: BOOLEAN);

  METHOD ReadIntDict (FileName: ARRAY OF CHAR): IntDictionary;

  ⟨Various constants 323b⟩
  ⟨Querying the status of a parser 324a⟩
  ⟨Deferred methods to be redefined in generated parsers 324b⟩

END BaseParser;
```

The states of the LR automaton of all parsers generated by `DURA` are numbered. The final state of the parser has a conventionally stated state number which we hope won't collide with the other states of the automaton. `FinalState`

323b ⟨*Various constants* 323b⟩≡
```
CONST
  FinalState = 100001;
```

The following constants define an enumeration, which define the various form of cut level attached to a reduction, defining its behaviour in case of backtracking. See 6.9.2 for a thorough explanation of the difference between full and snip cut.

323c ⟨*Various constants* 323b⟩+≡
```
CONST
  FullCut = 2;
  SnipCut = 1;
  DefaultCut = 0;
  NoCut = -1;
```

`Status` can be used to query the current state of the parser. The various values it can return, including various possible error codes, are listed in the constant defined hereunder.

324a ⟨*Querying the status of a parser* 324a⟩≡

```
DEFERRED METHOD Status: INTEGER;

CONST
  Parsing = 900;
  NoError = 901;
  RecoverableError = 902;
  UnrecoverableError = 903;
  StepLimitError = 904;
```

The following methods are meant to be redefined in actual parsers generated by DURA (See page 865).

324b ⟨*Deferred methods to be redefined in generated parsers* 324b⟩≡

```
DEFERRED METHOD Parse(Lexer: LexerClass): RootClass;
DEFERRED METHOD SetErrorLimit (TheLimit: INTEGER);
DEFERRED METHOD SetStepLimit (TheLimit: INTEGER);
DEFERRED METHOD IsFetchedAttribute (Attr: Attribute): BOOLEAN;
DEFERRED METHOD StatAcc: ParserStatAcc;
DEFERRED METHOD DumpRecoverTable (Output: OutputStream);
```

324c ⟨`BaseParser` *class implementation* 324c⟩≡

```
CLASS BaseParser (RootClass IN BaseNonTerminal,
                  LexerClass IN YLexicalAnalyzer);

  ⟨Dumping a non-terminal in a generic way, implementation 324d⟩
  ⟨Reading an IntDictionary, implementation 330b⟩

END BaseParser;
```

324d ⟨*Dumping a non-terminal in a generic way, implementation* 324d⟩≡

```
METHOD DumpNonTerminal (Output: OutputStream;
                        Nt: BaseNonTerminal;
                           Lex: YLexicalAnalyzer;
                        Full: BOOLEAN);

  ⟨LocalDump method 325⟩

  BEGIN
  LocalDump (Output, Nt, Full, 0);
  END DumpNonTerminal;
```

325 ⟨`LocalDump` *method* 325⟩≡

```
METHOD LocalDump(Output: OutputStream;
                  Nt: BaseNonTerminal;
                 Full: BOOLEAN;
                 Offset: INTEGER);
  VAR
    i, Ofs: INTEGER;
    FirstAttr: BOOLEAN;

  ⟨Attach Nt as the current instance of the metaclass 328b⟩
  ⟨IgnoreAttr local method 329a⟩
  ⟨OutputList local method 329b⟩
  ⟨NewLine local method 330a⟩

  BEGIN
  Ofs ← Offset;
  Output.WriteString (Nt.ClassName);
  Output.WriteString (“((”);
  Output.WriteInt (Nt.LineNr, 0);
  Output.WriteChar (“,”);
  Output.WriteInt (Nt.ColumnNr, 0);
  Output.WriteString (“),(”);
  Output.WriteInt (Nt.EndLineNr, 0);
  Output.WriteChar (“,”);
  Output.WriteInt (Nt.EndColumnNr, 0);
  Output.WriteChar (“)”);
  FirstAttr ← TRUE;
  ASSERT Nt.GetMetaClass ≠ Λ;
  SetCurrentInstance(Nt.GetMetaClass, Nt);
  ⟨Loop on all attributes 326⟩
  Output.WriteChar (“)”);
  END LocalDump;
```

After dumping the global information about the non-terminal `Nt`, we now on loop those of the attributes listed in its metaclass that we don't wish to skip (See method *DuraBase.BaseParser.IsFetchedAttribute* on page 324) and that do hold a significant value. `Ofs` holds the offset used to output subnodes, thereby making the tree structure clear.

326 ⟨*Loop on all attributes* 326⟩≡

```
FOR Attr ∈ Nt.GetMetaClass.AttributeList |
              IsFetchedAttribute(Attr) ∧
              (Full ∨ ¬ IgnoreAttr(Attr)) DO
  IF FirstAttr THEN
    Ofs ← Ofs + 2;
    FirstAttr ← FALSE;
    END;
  Output.WriteChar (‘‘,”);
  NewLine;
  Output.WriteString (Attr.Name);
  Output.WriteChar (‘‘=”);
  SetCurrentInstance(Nt.GetMetaClass, Nt);
  ⟨Output according to the kind of attribute 327⟩
  END;
```

First, we consider predefined types, including character strings.

327 ⟨*Output according to the kind of attribute* 327⟩≡

```
WHAT Attr OF
  IN IntegerAttribute:
    i ← TAG.Get;
    IF (i ≥ Lex.FIRSTTOKEN) ∧ (i ≤ Lex.LastToken) THEN
      Output.WriteString (Lex.TokenImage (i));
     ELSE
      Output.WriteInt (i, 0);
      END;
    END;
  IN BooleanAttribute:
    IF TAG.Get THEN
      Output.WriteString (“TRUE”);
     ELSE
      Output.WriteString (“FALSE”);
      END;
    END;
  IN StringAttribute:
    IF TAG.Get = Λ THEN
      Output.WriteString (“<void>”);
     ELSE
      Output.WriteChar (“'”);
      Output.WriteString (TAG.Get);
      Output.WriteChar (“'”);
      END;
    END;
  ⟨If the attribute refers to a class, special action is needed 328a⟩
  END;
```

Then, if it refers to a user-defined class, two cases must be taken into account:

- The attribute refers to another non-terminal, in which case we perform a recursive call to `LocalDump`, increasing the indentation level by 2.
- The attribute refers to a list of non-terminals, by inheriting from the `ArrayList` class as provided by the YAFL standard library.

328a ⟨*If the attribute refers to a class, special action is needed* 328a⟩≡

```
IN ClassAttribute:
  IF TAG.Get = Λ THEN
    Output.WriteString (“VOID”);
   ELSE
    WHAT TAG.Get OF
      IN BaseNonTerminal:
        LocalDump (Output, TAG, Full, Ofs + 2);
         END;
      IN ArrayList:
        Output.WriteChar (“[”);
        Ofs ← Ofs + 2;
        NewLine;
        OutputList (TAG);
        Ofs ← Ofs - 2;
        NewLine;
        Output.WriteChar (“]”);
        END;
      END;
    END;
  END;
```

YAFL

`SetCurrentInstance` attaches a non-terminal to its attached metaclass. By construction, we know that the metaclass matches the non-terminal's class, but we need to constrain the types as managed by the YAFL compiler before attaching the instance to its metaclass by using the `UseInstance` method.

We need a separate method to do this, because the ability to test a parameterized variable's actual class parameter is restricted to formals to methods. Hence, we must write a separate method that takes the `MetaClass` as parameter, instead of referring directly to a local variable.

328b ⟨*Attach* `Nt` *as the current instance of the metaclass* 328b⟩≡

```
METHOD SetCurrentInstance (MClass: MetaClass;
                           Nt: BaseNonTerminal);
  BEGIN
  WHAT Nt OF
    IN MClass.Class:
      MClass.UseInstance (TAG);
      END;
    END;
  END SetCurrentInstance;
```

`IgnoreAttr` indicates whether an attribute can be neglected based on its content, that is, the value of this attribute for the current instance attached to its enclosing metaclass. The criterion is type-dependant, and based on the attribute's value.

329a ⟨`IgnoreAttr` *local method* 329a⟩≡

```
METHOD IgnoreAttr (Attr: Attribute): BOOLEAN;
  BEGIN
  WHAT Attr OF
    IN IntegerAttribute:
      RESULT ← TAG.Get = 0;
      END;
    IN BooleanAttribute:
      RESULT ← ¬ TAG.Get;
      END;
    IN StringAttribute:
      RESULT ← TAG.Get = Λ;
      END;
    IN ClassAttribute:
      RESULT ← TAG.Get = Λ;
      END;
    END;
  END IgnoreAttr;
```

329b ⟨`OutputList` *local method* 329b⟩≡

```
METHOD OutputList (l: ArrayList);
  BEGIN
  FOR i ∈ 0 .. l.Size - 1 DO
    IF i > 0 THEN
      Output.WriteChar (",");
      NewLine;
      END;
    WHAT l_i OF
      IN BaseNonTerminal:
        LocalDump (Output, TAG, Full, Ofs);
        END;
      END;
    END;
  END OutputList;
```

Send a new line, taking current indentation level into account.

330a ⟨NewLine *local method* 330a⟩≡

```
METHOD NewLine;
  BEGIN
  Output.WriteLn;
  FOR i ∈ 1 .. Ofs DO
    Output.WriteChar (“ ”);
    END;
  END NewLine;
```

The `ReadIntDict` loads and returns an `IntDictionary` (See page 764). This method has been extracted from the skeleton because its behaviour is always the same, no matter the parser; and because it now allows redefinition, for instance, to allow the `IntDictionary` to be read from another directory or even from another file.

330b ⟨*Reading an* IntDictionary*, implementation* 330b⟩≡

```
METHOD ReadIntDict (FileName: ARRAY OF CHAR): IntDictionary;
  BEGIN
  RESULT.CREATE;
  IF ¬ RESULT.Load (FileName) THEN
    RESULT ← Λ;
    END;
  END ReadIntDict;
```

7.5.1 Elements of the LR stack

DURA uses a stack that matches quite closely the common kind of LALR stack as described extensively in the literature. There is a significant difference, however: unlike the good old LALR stack, all operations on the DURA stack for which backtracking has not been restricted (See method *DuraBase.ReduceStackElement.Cut* on page 344) must be reversible. Any direction that has been taken can be reversed.

Hence, whatever will be stored on the stack must include information for resynchronization if required.

As opposed to most `yacc` and `yacc`-like implementations that commonly use `C` as target language, we must cope with YAFL's strong typing. Hence, we must find ways of storing elements of different types on this stack without using any form of type casting.

We basically recognize two kinds of information that can be stored on this stack:

1. Shift transitions, that denote the fact that a given non-terminal has been consumed successfully. For instance, when accepting a keyword as part of a grammar rule, a shift transition entry is pushed onto our stack. This entry is marked with the keyword that has been accepted, as well as synchronization information referring to the state of the system before the keyword was recognized, in case backtracking must occur.

2. Reduce transitions, that denote the fact that one or more top entries of the `DURA` stack have been recognized as matching a non-terminal, that these entries have been removed from the stack, and replaced by this entry. For this reduce operation to be reversible, the reduce entry must keep track of the elements that had been removed from the stack in case the reduce operation must be undone.

This dichotomy between these two very different kinds of entries is implemented using three classes: one base class `ParserStackElement` that gathers all the common abilities among all entries, and resp. `ShiftStackElement` (See page 335) and `ReduceStackElement` (See page 337) to denote abilities specific to resp. shift and reduce entries.

$ParserStackElement$

$ParserStackElement$ serves as base class for the components $ShiftStackElement$ (See page 335) and $ReduceStackElement$ (See page 337) which are stored on the LR stack, as depicted extensively in the $Parser$ class (See page 240).

$FromState, ToState : \rho PDAState$ (See page 638)
$LexPosition, LexLength, LexCarnation : \mathbb{N}$
$ActionNr : \mathbb{N}$
$Sym : Symbol$ (See page 476)

$LexWeight$ is the number of tokens behind a $ParserStackElement$. Its value will be defined further in derived classes. $LexWeight$ is not implemented because it serves no purpose other than as an aid for the formal proof regarding the cases in which a `DURA`-generated parser can cycle (See page 5.5.1).

$LexWeight : \mathbb{N}$

$FromState \neq \Lambda$
$ToState \neq \Lambda$
$LexPosition \leq InputLength$ (See page 14)

331 ⟨*Elements to store on the evaluation stack, definition* 331⟩≡
⟨*Base class* `ParserStackElement` *definition* 332b⟩
⟨`ShiftStackElement` *class definition* 335⟩
⟨`ReduceStackElement` *class definition* 337⟩

332a ⟨*Elements to store on the evaluation stack, implementation* 332a⟩≡
⟨*Base class* ParserStackElement *implementation* 333a⟩
⟨ShiftStackElement *class implementation* 336a⟩
⟨ReduceStackElement *class implementation* 339⟩

332b ⟨*Base class* ParserStackElement *definition* 332b⟩≡

```
CLASS ParserStackElement;

   REDEFINE METHOD CREATE (FromStateNr,
                           ToStateNr,
                           ActionNr,
                           LexPos,
                           LexLength,
                           LexCarnation: INTEGER);


   ⟨State management, definition 334a⟩
   ⟨Position management, definition 333b⟩

END ParserStackElement;
```

333a ⟨*Base class* `ParserStackElement` *implementation* 333a⟩≡

```
CLASS ParserStackElement;

  VAR
    TheFromStateNr,
    TheToStateNr,
    TheActionNr,

    TheLexPos,
    TheLexLength,
    TheLexCarnation: INTEGER;

  REDEFINE METHOD CREATE (FromStateNr,
                          ToStateNr,
                          ActionNr,
                          LexPos,
                          LexLength,
                          LexCarnation: INTEGER);
    BEGIN
    BASE;
    TheFromStateNr ← FromStateNr;
    TheToStateNr ← ToStateNr;
    TheActionNr ← ActionNr;
    TheLexPos ← LexPos;
    TheLexLength ← LexLength;
    TheLexCarnation ← LexCarnation;
    END CREATE;

  ⟨State management, implementation 334b⟩

END ParserStackElement;
```

`FromPosition` and `ToPosition` are meant to be redefined by `ParserStackElement`-derived classes to return resp. the first and the last position of `StackElement`. When dealing with a shift, `FromPosition` and `ToPosition` returns resp. the position of the first and last character of the lexeme, while when dealing with a reduction, `FromPosition` returns the position of the beginning of the first lexeme, while `ToPosition` returns the last position of the last lexeme of the reduction.

333b ⟨*Position management, definition* 333b⟩≡

```
DEFERRED METHOD FromPosition: INTEGER;
DEFERRED METHOD ToPosition: INTEGER;
```

Whenever a `ParserStackElement` is pushed on the stack, the state of the system at that time is saved, so that if the operation must be undone, we can restore the system's state, and try another path of analysis.

334a ⟨*State management, definition* 334a⟩≡

```
METHOD LexCarnation: INTEGER;

METHOD ActionNr: INTEGER;
METHOD FromStateNr: INTEGER;
METHOD ToStateNr: INTEGER;
METHOD LexLength: INTEGER;
METHOD LexPos: INTEGER;
```

Well, of course, the corresponding implementation is not very interesting.

334b ⟨*State management, implementation* 334b⟩≡

```
METHOD LexLength: INTEGER;
  BEGIN
  RESULT ← TheLexLength;
  END LexLength;

METHOD LexPos: INTEGER;
  BEGIN
  RESULT ← TheLexPos;
  END LexPos;

METHOD LexCarnation: INTEGER;
  BEGIN
  RESULT ← TheLexCarnation;
  END LexCarnation;

METHOD FromStateNr: INTEGER;
  BEGIN
  RESULT ← TheFromStateNr;
  END FromStateNr;

METHOD ToStateNr: INTEGER;
  BEGIN
  RESULT ← TheToStateNr;
  END ToStateNr;

METHOD ActionNr: INTEGER;
  BEGIN
  RESULT ← TheActionNr;
  END ActionNr;
```

`ShiftStackElement` implements a shift operation on an LR-stack. The only information it holds, besides the state recovery attributes it inherits from `ParserStackElement` is the code on which the shift is performed.

ShiftStackElement

$ParserStackElement$ (See page 331)

$ActionrNr$ is always 1 in $ShiftStackElement$, since it is always the first action a parser considers if applicable.

$TheCode : TokenCode$ (See page 14)

$Sym = TheCode$
$ToState = \delta FromState.Transition(TheCode)$
$LexWeight = 1$
$ActionNr = 1$

335 ⟨`ShiftStackElement` *class definition* 335⟩≡

```
CLASS ShiftStackElement;
  INHERITS ParserStackElement;

  REDEFINE METHOD CREATE (FromStateNr,
                          ToStateNr,
                          ActionNr,
                          LexPos,
                          LexLength,
                          LexCarnation,
                          LexCode: INTEGER);

  METHOD LexCode: INTEGER;

END ShiftStackElement;
```

The code is implemented in the TheLexCode attribute, and can be consulted via the LexCode method.

336a ⟨ShiftStackElement *class implementation* 336a⟩≡

```
CLASS ShiftStackElement;
  INHERITS ParserStackElement;

  VAR
    TheLexCode: INTEGER;

  REDEFINE METHOD CREATE (FromStateNr,
                          ToStateNr,
                          ActionNr,
                          LexPos,
                          LexLength,
                          LexCarnation,
                          LexCode: INTEGER);
    BEGIN
    BASE (FromStateNr, ToStateNr,
          ActionNr,
          LexPos, LexLength, LexCarnation);
    TheLexCode ← LexCode;
    END CREATE;

  METHOD LexCode: INTEGER;
    BEGIN
    RESULT ← TheLexCode;
    END LexCode;

  ⟨Position management, as redefined by ShiftStackElement 336b⟩

END ShiftStackElement;
```

336b ⟨*Position management, as redefined by* ShiftStackElement 336b⟩≡

```
REDEFINE METHOD FromPosition: INTEGER;
  BEGIN
  RESULT ← LexPos;
  END FromPosition;

REDEFINE METHOD ToPosition: INTEGER;
  BEGIN
  RESULT ← LexPos + LexLength - 1;
  END ToPosition;
```

`ReduceStackElement` implements a reduce operation to be stored on the LR stack. Besides all the information common to all `ParserStackElements`, it includes the following additional information:

- An array of `ParserStackElement` which is the actual slice of the LR stack that was reduced, and which is kept in case one must be able to undo the reduce operation, and put this array back on the stack
- A RHS number, that indicates which production was used to actually perform the reduction
- A reference to a non-terminal, if the reduce is the top-level reduction that must be fetched into an attribute
- An indication whether the rule has been fetched into a non-terminal.

ReduceStackElement

$ParserStackElement$ (See page 331)

$TheRHS : \rho FlatRHS$ (See page 511)
$TheArray : \mathrm{seq} \downarrow ParserStackElement$

$Sym = \delta\, TheRHS.Enclosing$
$\#\, TheArray = \delta\, TheRHS.Length$

$LexWeight$ is defined as the sum of the weights of the elements of the reduction, and of course, it can only be null if the $FlatRule$ (See page 477) it refers to is at least nullable.

$LexWeight = \sum_{x \in \mathrm{ran}\ TheArray} x.LexWeight$
$LexWeight = 0 \Rightarrow (\delta(\delta\, TheRHS).Enclosing).NullableLevel = AlwaysNull \lor$
$\quad (\delta(\delta\, TheRHS).Enclosing).NullableLevel = Nullable$

337 ⟨`ReduceStackElement` *class definition* 337⟩≡

```
CLASS ReduceStackElement;
  INHERITS ParserStackElement;

  REDEFINE METHOD CREATE (FromStateNr,
                          ToStateNr,
                          ActionNr,
                          LexPos,
                          LexLength,
                          LexCarnation,
                          RHSNr: INTEGER;
                          Arr: ARRAY OF ParserStackElement);
  METHOD Array: ARRAY OF ParserStackElement;
    POST
      NO_CHANGE;
```

⟨*Cut handling, definition* 343c⟩

```
METHOD RHSNr: INTEGER;
  POST
    NO_CHANGE;
```

⟨*Setting and consulting the synthetized non-terminal, definition* 340b⟩

```
  METHOD IsFetched: BOOLEAN;
    POST
      NO_CHANGE;

  METHOD SetFetched;
    POST
      IsFetched;

END ReduceStackElement;
```

339 ⟨`ReduceStackElement` *class implementation* 339⟩≡

```
CLASS ReduceStackElement;
  INHERITS ParserStackElement;

  VAR
    TheArray: ARRAY OF ParserStackElement;
    TheRHSNr: INTEGER;

  REDEFINE METHOD CREATE (FromStateNr, ToStateNr,
                          ActionNr,
                          LexPos,
                          LexLength,
                          LexCarnation,
                          RHSNr: INTEGER;
                          Arr: ARRAY OF ParserStackElement);
    BEGIN
    BASE (FromStateNr,
          ToStateNr,
          ActionNr ← ActionNr,
          LexPos ← LexPos,
          LexLength ← LexLength,
          LexCarnation ← LexCarnation);
    TheRHSNr ← RHSNr;
    TheArray ← Arr;
    END CREATE;


  ⟨ReduceStackElement, consulting the attributes 340a⟩
  ⟨Setting and consulting the synthetized non-terminal, implementation 341⟩
  ⟨Position management as redefined by ReduceStackElement 342a⟩

  ⟨Cut handling 344⟩
  ⟨Decide whether the content of the reduction has been fetched 346b⟩

END ReduceStackElement;
```

340a ⟨ReduceStackElement, *consulting the attributes* 340a⟩≡

```
METHOD Array: ARRAY OF ParserStackElement;
  BEGIN
  RESULT ← TheArray;
  END Array;

METHOD RHSNr: INTEGER;
  BEGIN
  RESULT ← TheRHSNr;
  END RHSNr;
```

340b ⟨*Setting and consulting the synthetized non-terminal, definition* 340b⟩≡

```
METHOD NonTerminal: BaseNonTerminal;
  POST
    NO_CHANGE;

METHOD SetNonTerminal (NonTerminal: BaseNonTerminal);
METHOD SetNonTerminalNoConnect (NonTerminal: BaseNonTerminal);
METHOD DetachNonTerminal;
```

One can attach a non-terminal to a `ReduceStackElement`. The default for this link is to be bi-directional, that is, that the `BaseNonTerminal` refers to the `ReduceStackElement`.

However, this link can be made unidirectional, by using the `SetNonTerminalNoConnect` method which can be useful if the non-terminal already refers to another `ReduceStackElement` and if one wishes not to lose the existing link.

`DetachNonTerminal` breaks the relationship in one direction (namely, from the `BaseNonTerminal` onto the `ReduceStackElement`), but maintains it in the opposite direction because we might have to ask what the attached non-terminal is for a `ReduceStackElement` for which no backtracking can occur any longer.

341 ⟨*Setting and consulting the synthetized non-terminal, implementation* 341⟩≡

```
VAR
  TheNonTerminal: BaseNonTerminal;

METHOD SetNonTerminal (NonTerminal: BaseNonTerminal);
  BEGIN
  SetNonTerminalNoConnect (NonTerminal);
  TheNonTerminal.SetReduceStackElement (THIS);
  END SetNonTerminal;

METHOD SetNonTerminalNoConnect (NonTerminal: BaseNonTerminal);
  BEGIN
  TheNonTerminal ← NonTerminal;
  END SetNonTerminalNoConnect;

METHOD NonTerminal: BaseNonTerminal;
  BEGIN
  RESULT ← TheNonTerminal;
  END NonTerminal;

METHOD DetachNonTerminal;
  BEGIN
  IF TheNonTerminal ≠ Λ THEN
    TheNonTerminal.SetReduceStackElement (Λ);
    END;
  END DetachNonTerminal;
```

The starting position of a reduction is the starting position of the leftmost shift operation it contains. If one tries to use a fully recursive implementation, one quickly bumps into system limitations due to excessive recursion. This is a consequence of the implementation of iteration as left-recursion. Hence, finding the leftmost shift in an iteration of n elements (statements, for instance) requires a recursive plunge of n levels.

In order to cope with this problem, the following implementation first tries an iterative method to find the leftmost shift. If this method fails, for instance because the reduction is *not* the result of iteration, and if it is made of a null prefix, we fall back on the more intuitive recursive access method, which should not require excessive stack depth, since the abnormally recursive case should have been take care of.

342a ⟨*Position management as redefined by* ReduceStackElement 342a⟩≡

```
REDEFINE METHOD FromPosition: INTEGER;
  VAR
    GoOn: BOOLEAN;
    Current: ParserStackElement;
  BEGIN
  RESULT ← -1;
  -- Try left most...
  IF (TheArray ≠ Λ) ∧ (TheArray.SIZE > 0) THEN
    Current ← TheArray_0;
    GoOn ← TRUE;
    WHILE GoOn DO
      GoOn ← FALSE;
      WHAT Current OF
        IN ReduceStackElement:
          IF (TAG.TheArray ≠ Λ) ∧ (TAG.TheArray.SIZE > 0) THEN
            IF TAG.TheArray_0 ≠ Λ THEN
              Current ← TAG.TheArray_0;
              GoOn ← TRUE;
              END;
            END;
          END;
       ELSE
        END;
      END;
    RESULT ← Current.FromPosition;
    IF RESULT < 0 THEN
      ⟨Use naive recursive method 342b⟩
      END;
    END;
  ⟨If required, check attached non-terminal 343a⟩
  END FromPosition;
```

Of course, the naive version is far easier to understand than the derecursified one.

342b ⟨*Use naive recursive method* 342b⟩≡

```
FOR x ∈ TheArray WHILE RESULT < 0 DO
  RESULT ← x.FromPosition;
  END;
```

If nothing else seems to work, we also consider the possibility of asking this position information to the attached non-terminal.

343a ⟨*If required, check attached non-terminal* 343a⟩≡
```
IF (RESULT < 0) ∧ (TheNonTerminal ≠ Λ) THEN
   RESULT ← TheNonTerminal.FromPosition;
   END;
```

Since a LR automaton uses left-recursion to represent iteration, there is no need for any derecursification technique for the rightmost position. The simple recursive method works fine.

Just as for the `FromPosition` described above, if `TheArray` is VOID, we also consider the possibility of getting the position information from the attached non-terminal.

343b ⟨*Position management as redefined by* `ReduceStackElement` 342a⟩+≡
```
REDEFINE METHOD ToPosition: INTEGER;
   BEGIN
   RESULT ← -1;
   IF TheArray ≠ Λ THEN
      FOR i ∈ TheArray.SIZE - 1 .. 0 BY -1 WHILE RESULT < 0 DO
         RESULT ← TheArray_i.ToPosition;
         END;
    ELSIF (TheNonTerminal ≠ Λ) ∧
           (TheNonTerminal.ToPosition > 0) THEN
      RESULT ← TheNonTerminal.ToPosition;
      END;
   END ToPosition;
```

343c ⟨*Cut handling, definition* 343c⟩≡
```
METHOD Cut (Level: INTEGER);
   PRE
    (Level = BaseParser.FullCut) ∨
    (Level = BaseParser.SnipCut) ∨
    (Level = BaseParser.NoCut);
   POST
      CutLevel = Level;

METHOD Snip;
   POST
      CutLevel = BaseParser.SnipCut;

METHOD CutLevel: INTEGER;
   POST
    NO_CHANGE,
    (RESULT = BaseParser.FullCut) ∨
    (RESULT = BaseParser.SnipCut) ∨
    (RESULT = BaseParser.NoCut);
```

Cut handling by the `ReduceStackElement` class is a double matter:

- As long as the reduce operation can be undone, the array that has been removed from the LR stack must be kept in case it must be unreduced, that is, put back onto the LR stack.
- A status is maintained in `TheCutLevel` to distinguish between snips, that denote reductions that cannot be undone but that can be skipped when backtracking, and full cuts, that cancel the parsing process altogether when backtracked.

344 ⟨*Cut handling* 344⟩≡

```
VAR
  TheCutLevel: INTEGER;

METHOD CutLevel: INTEGER;
  BEGIN
  RESULT ← TheCutLevel;
  END CutLevel;

METHOD Cut(Level: INTEGER);
  VAR
    Current, Next: ReduceStackElement;
    Arr: ARRAY OF ParserStackElement;
  BEGIN
  IF (Level = BaseParser.FullCut) ∨
     (Level = BaseParser.SnipCut) THEN
    ⟨Undo the entire backtracking structure 345a⟩
    END;
  TheCutLevel ← Level;
  END Cut;
```

The process here is slightly derecursified. We maintain `Next` that serves as a single element stack which indicates the following `ReduceStackElement` to visit. Recursive plunge is only required when `Next` already holds a value.

For the rest, cleaning up the backtracking structure is not a very complex process. We merely remove as many references as we can, so that memory can be reclaimed by the YAFL garbage collector. We use `DetachNonTerminal` (See page 341) to remove the bi-directional link to the attached non-terminal, clear the reference to the array, and even, clear the contents of the array itself.

345a ⟨*Undo the entire backtracking structure* 345a⟩≡

```
Current ← THIS;
WHILE Current ≠ Λ DO
  Next ← Λ;
  Current.DetachNonTerminal;
  Current.TheCutLevel ← Level;
  Arr ← Current.TheArray;
  IF Arr ≠ Λ THEN
    FOR El ∈ Arr | El ≠ Λ DO
      WHAT El OF
        IN ReduceStackElement:
          IF Next = Λ THEN
            Next ← TAG;
           ELSE
            TAG.Cut(Level);
            END;
          END;
       ELSE
        END;
      END;
    ⟨Reset Arr 345b⟩
    END;
  Current.TheArray ← Λ;
  Current ← Next;
  END;
```

Filling an array with `VOID` is optimized by the YAFL compiler (See page 973).

345b ⟨*Reset* `Arr` 345b⟩≡

```
FOR i ∈ 0 .. Arr.SIZE - 1 DO
  Arr_i ← Λ;
  END;
```

The following methods are mere conveniency methods around `Cut` (See page 344).

346a ⟨*Cut handling* 344⟩+≡
```
METHOD Snip;
  BEGIN
  Cut (BaseParser.SnipCut);
  END Snip;

METHOD FullCut;
  BEGIN
  Cut (BaseParser.FullCut);
  END FullCut;
```

The following chunk maintains the information about the `ReduceStackElement` being fetched into attributes of an object. Whenever the `FetchAttributes` method (See page 884) fetches attributes from a reduction, it also attempts to fetch subrules. It marks the `ReduceStackElement` as being fetched to prevent any further attempt to fetch attributes from this reduction.

346b ⟨*Decide whether the content of the reduction has been fetched* 346b⟩≡
```
VAR
  TheFetched: BOOLEAN;

METHOD IsFetched: BOOLEAN;
  BEGIN
  RESULT ← TheFetched;
  END IsFetched;

METHOD SetFetched;
  BEGIN
  TheFetched ← TRUE;
  END SetFetched;
```

`BaseNonTerminal` is the base class for all non-terminal classes that are defined within a DURA system. It provides various generic services, that apply to all non-terminals alike.

346c ⟨`BaseNonTerminal` *class definition* 346c⟩≡
```
CLASS BaseNonTerminal;

  ⟨Commit and RollBack, definition 347b⟩
  ⟨Attaching a metaclass to a non-terminal, definition 348a⟩
  ⟨Position handling, definition 349a⟩

  ⟨Attaching a ReduceStackElement to a non-terminal, definition 354c⟩

  ⟨Hierarchy handling, definition 356⟩

END BaseNonTerminal;
```

347a ⟨`BaseNonTerminal` *class implementation* 347a⟩≡
```
  CLASS BaseNonTerminal;

    ⟨Commit and RollBack, implementation 347c⟩
    ⟨Attaching a metaclass to a non-terminal, implementation 348b⟩
    ⟨Position handling, implementation 349b⟩
    ⟨Attaching a ReduceStackElement to a non-terminal, implementation 355a⟩
    ⟨Hierarchy handling, implementation 357b⟩

    END BaseNonTerminal;
```

`Commit` is called after reducing a non-terminal. A derived class can redefine this method to add some processing at creation time. Besides, if `Commit` returns `FALSE`, the reduction is cancelled, and another way of understanding the input is searched for.

This allows some non-syntactical property to be checked at parse time, rejecting a reduction if it does not fulfill the property. For instance, explains how checking that the closing identifier of a Modula II procedure, as in

```
PROCEDURE Example;
  BEGIN
  ...
  END Example;
```

matches its header cannot be done using context-free grammars only. Such a check could be integrated in a redefinition of a `Commit` method for procedure implementations, and forcing the parser to backtrack to find another way of understanding the input if the closing identifier does not match the header.

`RollBack` is called when the reduction attached to a non-terminal is unreduced during backtracking. One can redefine this method to perform tasks such as removing an element from a symbol table. It is however important to note that, due to implementation restrictions, `RollBack` will *not* be called if the non-terminal is enclosed within a snipped reduction.

347b ⟨`Commit` *and* `RollBack`*, definition* 347b⟩≡
```
  METHOD Commit: BOOLEAN;
  METHOD RollBack;
```

Given the explanation given above, the default implementations for `Commit` and `RollBack` are rather predictable.

347c ⟨`Commit` *and* `RollBack`*, implementation* 347c⟩≡
```
  METHOD Commit: BOOLEAN;
    BEGIN
    RESULT ← TRUE;
    END Commit;

  METHOD RollBack;
    BEGIN
    END RollBack;
```

Every non-terminal is attached to its corresponding meta-class when created by the parser (See method *DuraClDefinition.DuraClassDefinition.GenerateCreationMethodBody* on page 431).

348a ⟨*Attaching a metaclass to a non-terminal, definition* 348a⟩≡

```
METHOD SetMetaClass (MClass: MetaClass);
METHOD GetMetaClass: MetaClass;
METHOD ClassName: ARRAY OF CHAR;
```

348b ⟨*Attaching a metaclass to a non-terminal, implementation* 348b⟩≡

```
VAR
  TheMetaClass: MetaClass;

METHOD SetMetaClass (MClass: MetaClass);
  BEGIN
  TheMetaClass ← MClass;
  END SetMetaClass;

METHOD GetMetaClass: MetaClass;
  BEGIN
  RESULT ← TheMetaClass;
  END GetMetaClass;

METHOD ClassName: ARRAY OF CHAR;
  BEGIN
  IF TheMetaClass ≠ Λ THEN
    RESULT ← TheMetaClass.Name;
    END;
  END ClassName;
```

7.5.2 Position handling

Position handling for non-terminals is divided in two different groups of methods:

- Methods which deal with positions expressed in terms of offsets from the beginning of the file. Such positions, implemented as integers, are convenient to deal with from a computing point of view, but hard to manage for the human reader.
- Methods which deal with positions expressed in terms of line and column numbers, much easier to deal with for the human reader.

Let's first see how things work when dealing with integer positions.

Each non-terminal has a starting position `FromPosition` and an ending position `ToPosition`, both defined as integer offsets relative to the beginning of the input file.

349a ⟨*Position handling, definition* 349a⟩≡

```
METHOD SetPositions(FromPos, ToPos: INTEGER);
  PRE
    ToPos ≥ FromPos;
  POST
    FromPosition = FromPos,
    ToPosition = ToPos;

METHOD FromPosition: INTEGER;
  POST
    NO_CHANGE;

METHOD ToPosition: INTEGER;
  POST
    NO_CHANGE;
```

349b ⟨*Position handling, implementation* 349b⟩≡

```
VAR
  TheFromPos, TheToPos: INTEGER;

METHOD SetPositions(FromPos, ToPos: INTEGER);
  BEGIN
  TheFromPos ← FromPos;
  TheToPos ← ToPos;
  END SetPositions;

METHOD FromPosition: INTEGER;
  BEGIN
  RESULT ← TheFromPos;
  END FromPosition;

METHOD ToPosition: INTEGER;
  BEGIN
  RESULT ← TheToPos;
  END ToPosition;
```

Then, one can use `ComputeLineAndColNr` to turn these positions into line and column numbers. `RecursiveComputeLineAndColNr` performs the same operation repeatedly on the node and all its subnodes.

350a ⟨*Position handling, definition* 349a⟩+≡

```
METHOD ComputeLineAndColNr (CharStr: AbstractCharStream);
METHOD RecursiveComputeLineAndColNr
                                  (CharStr: AbstractCharStream);
```

Basically, turning an offset-based position into a line and column number position is performed by successively setting the current position of the character stream (See page 190) to the starting and ending position of the non-terminal, invoking `ComputeLineAndColumnNr`, as defined by `AbstractCharStream` (See page 192) and storing the computed line and column numbers in ad hoc attributes.

350b ⟨*Position handling, implementation* 349b⟩+≡

```
VAR
  TheLineNr,
  TheColumnNr,
  TheEndLineNr,
  TheEndColumnNr: INTEGER;

METHOD ComputeLineAndColNr (CharStr: AbstractCharStream);
  BEGIN
  IF (FromPosition ≥ 0) ∧ (ToPosition ≥ FromPosition) THEN
    CharStr.SetPosition (FromPosition);
    CharStr.ComputeLineAndColumnNr(SYSTEM.CHR (SYSTEM.NewLine));
    THIS.TheLineNr ← CharStr.LineNr;
    THIS.TheColumnNr ← CharStr.ColumnNr;
    CharStr.SetPosition (THIS.ToPosition);
    CharStr.ComputeLineAndColumnNr(SYSTEM.CHR (SYSTEM.NewLine));
    THIS.TheEndLineNr ← CharStr.LineNr;
    THIS.TheEndColumnNr ← CharStr.ColumnNr;
    END;
  END ComputeLineAndColNr;
```

The recursive version of `ComputeLineAndColNr` navigates through the non-terminal's siblings (See method *DuraBase.BaseNonTerminal.Sibling* on page 357) and next references (See method *DuraBase.BaseNonTerminal.Next* on page 356) and computes the line and column number for each of the non-terminals it encounters.

351a ⟨*Position handling, implementation* 349b⟩+≡

```
METHOD RecursiveComputeLineAndColNr
                         (CharStr: AbstractCharStream);
  VAR
    r, s: BaseNonTerminal;
  BEGIN
  s ← THIS;
  WHILE s ≠ Λ DO
    r ← s;
    WHILE r ≠ Λ DO
      IF ⟨r's line number has not been computed 351b⟩ THEN
        r.ComputeLineAndColNr (CharStr);
        IF r.FirstSon ≠ Λ THEN
          r.FirstSon.RecursiveComputeLineAndColNr (CharStr);
          END;
        END;
      r ← r.Next;
      ⟨Test for a node with two paths from its father 352⟩
      END;
    s ← s.Sibling;
    END;
  END RecursiveComputeLineAndColNr;
```

We assume that line numbering starts at 1, so a non-terminal with line number negative or null does not have a valid line number yet.

351b ⟨*r's line number has not been computed* 351b⟩≡

```
r.TheLineNr ≤ 0
```

Normally, the `Next` pointer is used to connect elements of sequences built after a repetition operator "+" or "*", while the `FirstSon` and `Sibling` pointers implement the attributes of an object. In some case, a non-terminal can be reached from the two paths. For instance, consider the following extract of the `COBOL` grammar:

```
GRAMMAR
  ALevelImage := INumber
  (TheId OR (TheIsFiller := FillerKeyword)) ?
      (
        ThePicClause OR TheValueClause OR
        TheUsageClause OR TheSyncClause OR
        TheJustClause  OR
        TheSignClause OR
        TheRenamesClause OR
        TheOccursClause OR
        TheSourceClause OR
        TheObjectClause
      )*
    Dot;

VAR
  ALevelImage: ARRAY OF CHAR;
  TheIsFiller: BOOLEAN;
  TheId: DataIdent;
  ThePicClause: PicClause;
  TheValueClause: ValueClause;
  TheUsageClause: UsageClause;
  TheJustClause: JustClause;
  TheRenamesClause: RenamesClause;
  TheSignClause: SignClause;
  TheSourceClause: SourceClause;
  TheObjectClause: ObjectClause;
  TheOccursClause: OccursClause;
```

The purpose of this grammar rule is to state that the various subcomponents of a `COBOL` data field can be specified in any order. Of course, this grammar rule will parse things that are invalid, such as a variable declaration with more than one `PICTURE` clause. We assume such cases to be highly unlikely.

The values of the attributes `ThePicClause`, `TheValueClause`, etc. are accessible starting at the non-terminal that denotes the data field using the `FirstSon` ans `Sibling` chain of references. However, since they part of a repetition operator ("*"), they are also connected by the `Next` pointers. The following chunk detects such cases, and prevents the line numbers to be computed more than once if r, reached through the `Next` pointers, is also reachable as a direct attribute (See method *DuraBase.BaseNonTerminal.IsDirectSon* on page 359).

352 ⟨*Test for a node with two paths from its father* 352⟩≡

```
IF (r ≠ Λ) ∧ (r.Father ≠ Λ) ∧
   (r.Father.IsDirectSon(r)) THEN
  r ← Λ;
  END;
```

353a ⟨*Position handling, definition* 349a⟩+≡

```
  METHOD LineNr: INTEGER;
    POST
      NO_CHANGE;

  METHOD ColumnNr: INTEGER;
    POST
      NO_CHANGE;

  METHOD EndLineNr: INTEGER;
    POST
      NO_CHANGE,
      RESULT ≥ LineNr;

  METHOD EndColumnNr: INTEGER;
    POST
      NO_CHANGE,
      (LineNr = EndLineNr) ⇒ RESULT ≥ ColumnNr;
```

After that the line and column numbers have been computed, querying the attributes is straightforward.

353b ⟨*Position handling, implementation* 349b⟩+≡

```
  METHOD LineNr: INTEGER;
    BEGIN
    RESULT ← TheLineNr;
    END LineNr;

  METHOD ColumnNr: INTEGER;
    BEGIN
    RESULT ← TheColumnNr;
    END ColumnNr;

  METHOD EndLineNr: INTEGER;
    BEGIN
    RESULT ← TheEndLineNr;
    END EndLineNr;

  METHOD EndColumnNr: INTEGER;
    BEGIN
    RESULT ← TheEndColumnNr;
    END EndColumnNr;
```

In some applications, we need to be able to adapt the position of a non-terminal, taking margins or preprocessing into account. The two following methods provide us with the ability to arbitrarily change a non-terminal's position.

354a ⟨*Position handling, definition* 349a⟩+≡

```
METHOD SetLineAndColNr (ALineNr, AColNr: INTEGER);
  POST
    ALineNr = LineNr,
    AColNr = ColumnNr;

METHOD SetEndLineAndColNr (ALineNr, AColNr: INTEGER);
  POST
    ALineNr = EndLineNr,
    AColNr = EndColumnNr;
```

354b ⟨*Position handling, implementation* 349b⟩+≡

```
METHOD SetLineAndColNr (ALineNr, AColNr: INTEGER);
  BEGIN
  TheLineNr ← ALineNr;
  TheColumnNr ← AColNr;
  END SetLineAndColNr;

METHOD SetEndLineAndColNr (ALineNr, AColNr: INTEGER);
  BEGIN
  TheEndLineNr ← ALineNr;
  TheEndColumnNr ← AColNr;
  END SetEndLineAndColNr;
```

When reducing the outermost `ReduceStackElement` (See page 337) related to the synthesis of a non-terminal, the two components (that is, the `ReduceStack-Element` and the non-terminal) are connected to each other so that some operations that are performed on one of them can be reflected onto the other. For instance, calling `Cut` (See below) on a non-terminal basically means calling `Cut` onto the attached `ReduceStackElement`.

The following methods maintain this relationship, and provide several convenient shortcuts to trigger operations on the `ReduceStackElement` attached to a non-terminal.

354c ⟨*Attaching a* `ReduceStackElement` *to a non-terminal, definition* 354c⟩≡

```
METHOD SetReduceStackElement (StackEl: ReduceStackElement);
METHOD GetReduceStackElement: ReduceStackElement;
METHOD Cut(Level: INTEGER);
METHOD Snip;
METHOD FullCut;
```

When attaching a `ReduceStackElement` to a `BaseNonTerminal` for the first time (that is, to a `BaseNonTerminal` that has no attached `ReduceStackElement` yet), we copy the reduction's position information into the non-terminal by calling `SetPositions` (See page 349).

355a ⟨*Attaching a* `ReduceStackElement` *to a non-terminal, implementation* 355a⟩≡

```
VAR
  TheReduceStackElement: ReduceStackElement;

METHOD SetReduceStackElement (StackEl: ReduceStackElement);
  BEGIN
  IF (StackEl ≠ Λ) ∧
      (TheReduceStackElement = Λ) ∧
      (FromPosition ≤ 0) ∧
      (ToPosition ≤ 0) THEN
    SetPositions (StackEl.FromPosition,
                  StackEl.ToPosition);
    END;
  TheReduceStackElement ← StackEl;
  END SetReduceStackElement;

METHOD GetReduceStackElement: ReduceStackElement;
  BEGIN
  RESULT ← TheReduceStackElement;
  END GetReduceStackElement;
```

The following method are simply convenience shortcuts to allow us to control the backtracking level of a non-terminal, even if the actual processing happens at the attached `ReduceStackElement`'s level.

355b ⟨*Attaching a* `ReduceStackElement` *to a non-terminal, implementation* 355a⟩+≡

```
METHOD Cut(Level: INTEGER);
  BEGIN
  IF TheReduceStackElement ≠ Λ THEN
    TheReduceStackElement.Cut (Level);
    END;
  END Cut;

METHOD FullCut;
  BEGIN
  Cut(BaseParser.FullCut);
  END FullCut;

METHOD Snip;
  BEGIN
  Cut(BaseParser.SnipCut);
  END Snip;
```

7.5.3 Hierarchy handling

Hierarchy handling manages the various relationships between non-terminals. These relationships include:

- A `Father` relationship, that yields the closest enclosing non-terminal
- A `FirstSon` and `Sibling` relationship, that gives resp. the first non-terminal attached to a father, and then, successively, all the first son's siblings. The order in which these siblings will be delivered is not deterministic.
- When one of a node's siblings is the first member of a list (typically using one of DURA's repetition operators "+" or "*"), the other elements of the list can be fetched using the `Next` method.

356 ⟨*Hierarchy handling, definition* 356⟩≡

```
METHOD SetFather (TheFather: BaseNonTerminal);
  PRE
    Father = Λ;
  POST
    Father = TheFather;

METHOD Father: BaseNonTerminal;
  POST
    NO_CHANGE;

METHOD IsDirectSon (Nt: BaseNonTerminal): BOOLEAN;
  PRE
    Nt ≠ Λ;

METHOD FirstSon: BaseNonTerminal;
  POST
    NO_CHANGE;

METHOD SetPrevious (ThePrevious: BaseNonTerminal);
  PRE
    ThePrevious ≠ Λ;

METHOD Previous: BaseNonTerminal;

METHOD Sibling: BaseNonTerminal;
  POST
    NO_CHANGE,
    RESULT ≠ Λ ⇒ RESULT.Father = Father;

METHOD Next: BaseNonTerminal;
```

The following two methods are mere conveniency methods that shield the above methods by making sure their preconditions is fulfilled.

357a ⟨*Hierarchy handling, definition* 356⟩+≡

```
METHOD SetSon (Son: BaseNonTerminal);

METHOD SetSons (Sons: NonTerminalList);
```

The first methods are pretty straightforward... the only trick here is the lack of `SetNext` method, which action is implied by `SetPrevious`.

357b ⟨*Hierarchy handling, implementation* 357b⟩≡

```
VAR
  TheSibling,
  TheNext,
  ThePrevious,
  TheFirstSon,
  TheFather: BaseNonTerminal;

METHOD SetFather (TheFather: BaseNonTerminal);
  BEGIN
  THIS.TheFather ← TheFather;
  END SetFather;

METHOD Father: BaseNonTerminal;
  BEGIN
  RESULT ← TheFather;
  END Father;

METHOD SetPrevious (ThePrevious: BaseNonTerminal);
  BEGIN
  THIS.ThePrevious ← ThePrevious;
  IF ThePrevious ≠ Λ THEN
    ThePrevious.TheNext ← THIS;
    END;
  END SetPrevious;

METHOD Previous: BaseNonTerminal;
  BEGIN
  RESULT ← ThePrevious;
  END Previous;
```

358 ⟨*Hierarchy handling, implementation* 357b⟩+≡

```
METHOD Next: BaseNonTerminal;
  BEGIN
  RESULT ← TheNext;
  END Next;

METHOD SetSon (Son: BaseNonTerminal);
  BEGIN
  IF Son ≠ Λ THEN
    IF ¬ IsDirectSon(Son) THEN
      Son.TheSibling ← FirstSon;
      TheFirstSon ← Son;
      Son.SetFather (THIS);
     ELSE
      StdOut.WriteLine (“Already in sons ! ” +
                        ClassName + “ -> ” + Son.ClassName);
      END;
    END;
  END SetSon;

METHOD SetSons (Sons: NonTerminalList);
  BEGIN
  IF (Sons ≠ Λ) ∧ (Sons.Size > 0) THEN
    SetSon (Sons.First);
    FOR i ∈ 1 .. Sons.Size - 1 DO
      Sons_i.SetFather (THIS);
      END;
    END;
  END SetSons;
```

359a ⟨*Hierarchy handling, implementation* 357b⟩+≡

```
METHOD FirstSon: BaseNonTerminal;
  BEGIN
  RESULT ← TheFirstSon;
  END FirstSon;

METHOD Sibling: BaseNonTerminal;
  BEGIN
  RESULT ← TheSibling;
  END Sibling;

METHOD IsDirectSon (Nt: BaseNonTerminal): BOOLEAN;
  VAR
    f: BaseNonTerminal;
  BEGIN
  f ← FirstSon;
  WHILE (f ≠ Λ) ∧ (f ≠ Nt) DO
    ASSERT f.Father = THIS;
    ASSERT f.Sibling ≠ f;
    f ← f.Sibling;
    END;
  RESULT ← f ≠ Λ;
  END IsDirectSon;
```

The methods listed here bypass the various checks that are implemented in the methods listed above. They are used for low-level operations on the parse tree, such as removing an intermediate non-terminal, replacing references to it by references to its children.

359b ⟨*Hierarchy handling, definition* 356⟩+≡

```
METHOD ResetFirstSon (Nt: BaseNonTerminal);
  POST
    FirstSon = Λ;

METHOD RemoveSon(Son: BaseNonTerminal);

METHOD ResetSibling;
  POST
    Sibling = Λ;
```

Besides the fact that they bypass all pre-conditions and possible consistency checks, these methods are otherwise pretty straightforward.

360a ⟨*Hierarchy handling, implementation* 357b⟩+≡
```
METHOD RemoveSon(Son: BaseNonTerminal);
  VAR
    tmp: BaseNonTerminal;
  BEGIN
  IF Son ≠ Λ THEN
    IF TheFirstSon = Son THEN
      TheFirstSon ← Son.TheSibling;
    ELSE
      tmp ← TheFirstSon;
      WHILE (tmp ≠ Λ) ∧
            (tmp.TheSibling ≠ Son) ∧
            (tmp.TheSibling ≠ TheFirstSon) DO
        tmp ← tmp.TheSibling;
        END;
      IF (tmp ≠ Λ) ∧ (tmp.TheSibling = Son) THEN
        tmp.TheSibling ← Son.TheSibling;
        END;
      END;
    END;
  END RemoveSon;

METHOD ResetFirstSon (Nt: BaseNonTerminal);
  BEGIN
  TheFirstSon ← Nt;
  END ResetFirstSon;

METHOD ResetSibling;
  BEGIN
  TheSibling ← Λ;
  END ResetSibling;
```

NonTerminalList serves as base list for all collections in DURA-generated parsers. It inherits from ArrayList (provided by the YAFL standard library) which provides better performance and a smaller memory footprint than true linked lists, except when one performs lots of random insertion and deletion, but that should not be the case here.

360b ⟨NonTerminalList *class definition* 360b⟩≡
```
CLASS NonTerminalList(Element IN BaseNonTerminal);
  INHERITS ArrayList(Element);

END NonTerminalList;
```

361a ⟨NonTerminalList *class implementation* 361a⟩≡
```
CLASS NonTerminalList(Element IN BaseNonTerminal);
  INHERITS ArrayList(Element);

END NonTerminalList;
```

ParserStatAcc is a statistics accumulator, that encapsulates various informations gathered when running a DURA-generated parser, that can be combined in various ways and listed onto an output stream.

361b ⟨ParserStatAcc *class definition* 361b⟩≡
```
CLASS ParserStatAcc;

  METHOD Dump (Output: OutputStream;
               Verbose: BOOLEAN);
  METHOD ToString: ARRAY OF CHAR;

  METHOD Add(Other: ParserStatAcc);

  METHOD SetLineCount (LineCount: INTEGER);
  METHOD SetErrorCount (Count: INTEGER);
  METHOD SetLexicalBacktrackingCount (Count: INTEGER);
  METHOD SetShiftCounts (ShiftCount,
                         UnShiftCount: INTEGER);
    PRE
      ShiftCount ≥ UnShiftCount;

  ⟨SetReduceCounts, definition 364b⟩

  METHOD ReduceCount: INTEGER;
    POST
      NO_CHANGE;

  METHOD UnReduceCount: INTEGER;
    POST
      NO_CHANGE;

END ParserStatAcc;
```

362a ⟨ParserStatAcc *class implementation* 362a⟩≡

```
CLASS ParserStatAcc;

VAR
  TheErrorCount,
  TheLineCount,
  TheShiftCount,
  TheUnShiftCount,
  TheLexicalBacktrackingCount: INTEGER;
  TheRHSToRuleArr,
  TheReduceCounts,
  TheUnReduceCounts: ARRAY OF INTEGER;

  ⟨Adding a ParserStatAcc to another 362b⟩
  ⟨Dumping a ParserStatAcc 363a⟩
  ⟨Setting a ParserStatAcc's attributes 364a⟩
  ⟨SetReduceCounts, implementation 364c⟩
  ⟨Consulting synthetic information 365a⟩

END ParserStatAcc;
```

Add provides the ability to add a ParserStatAcc to another, to summarize the statistics of several runs of a DURA parser.

362b ⟨*Adding a* ParserStatAcc *to another* 362b⟩≡

```
METHOD Add(Other: ParserStatAcc);
  BEGIN
  TheLineCount ← TheLineCount + Other.TheLineCount;
  TheShiftCount ← TheShiftCount + Other.TheShiftCount;
  TheUnShiftCount ← TheUnShiftCount + Other.TheUnShiftCount;
  TheLexicalBacktrackingCount ←
    TheLexicalBacktrackingCount + Other.TheLexicalBacktrackingCount;
  ASSERT Other.TheReduceCounts.SIZE = TheReduceCounts.SIZE;
  ASSERT Other.TheUnReduceCounts.SIZE = TheUnReduceCounts.SIZE;
  FOR i ∈ 0 .. TheReduceCounts.SIZE - 1 DO
    TheReduceCounts_i ← TheReduceCounts_i +
                                    Other.TheReduceCounts_i;
    TheUnReduceCounts_i ← TheUnReduceCounts_i +
                                    Other.TheUnReduceCounts_i;
    END;
  END Add;
```

Turning a `ParserStatAcc` into a character string is rather predictable.

363a ⟨*Dumping a* `ParserStatAcc` 363a⟩≡

```
METHOD ToString : ARRAY OF CHAR;

  METHOD IToStr (i: INTEGER): ARRAY OF CHAR;
    BEGIN
    RESULT ← IntConversions.IntToString (i, 0);
    END IToStr;

  BEGIN
  RESULT ← IToStr(TheLineCount) +    " lin  " +
           IToStr(TheErrorCount) +   " err  " +
           IToStr(TheShiftCount) +   " shift  " +
           IToStr(TheUnShiftCount) + " unshift  " +
           IToStr(ReduceCount) +     " red  " +
           IToStr(UnReduceCount) +   " unred  " +
           IToStr(TheLexicalBacktrackingCount) +  " lexb";
  END ToString;
```

Based on `ToString` described above, actually dumping a `ParserStatAcc` onto an `OutputStream` is rather predictable. The only trick here is to use `TheRHSToRuleArr` that implements a function which translates a right hand side number into the number of the rule it belongs to. `TheRHSToRuleArr` is set by `SetReduceCounts` (See the following page).

363b ⟨*Dumping a* `ParserStatAcc` 363a⟩+≡

```
METHOD Dump (Output: OutputStream;
             Verbose: BOOLEAN);
  BEGIN
  Output.WriteLine (ToString);
  IF Verbose THEN
    FOR i ∈ 0 .. TheReduceCounts.SIZE - 1 |
          TheReduceCounts_i > 0 DO
      Output.WriteInt (TheRHSToRuleArr_i, 0);
      Output.WriteString (".");
      Output.WriteInt (i, 0);
      Output.WriteString (" = ");
      Output.WriteInt (TheReduceCounts_i, 0);
      Output.WriteString (".");
      Output.WriteInt (TheUnReduceCounts_i, 0);
      Output.WriteLn;
      END;
    END;
  END Dump;
```

364a ⟨*Setting a* `ParserStatAcc`*'s attributes* 364a⟩≡

```
METHOD SetLineCount (LineCount: INTEGER);
  BEGIN
  TheLineCount ← LineCount;
  END SetLineCount;

METHOD SetErrorCount (Count: INTEGER);
  BEGIN
  TheErrorCount ← Count;
  END SetErrorCount;

METHOD SetLexicalBacktrackingCount (Count: INTEGER);
  BEGIN
  TheLexicalBacktrackingCount ← Count;
  END SetLexicalBacktrackingCount;

METHOD SetShiftCounts (ShiftCount,
                       UnShiftCount: INTEGER);
  BEGIN
  TheShiftCount ← ShiftCount;
  TheUnShiftCount ← UnShiftCount;
  END SetShiftCounts;
```

This method's only interesting part is the extensive pre-condition that ensures that the various values it receives are consistent.

364b ⟨`SetReduceCounts`, *definition* 364b⟩≡

```
METHOD SetReduceCounts (ReduceCounts, UnReduceCounts,
                        RHSToRuleArr: ARRAY OF INTEGER);
  PRE
    ReduceCounts ≠ Λ,
    UnReduceCounts ≠ Λ,
    ReduceCounts.SIZE = UnReduceCounts.SIZE,
    ReduceCounts.SIZE = RHSToRuleArr.SIZE,
    ∀ i ∈ 0 .. ReduceCounts.SIZE - 1 •
      UnReduceCounts_i ≤ ReduceCounts_i;
```

364c ⟨`SetReduceCounts`, *implementation* 364c⟩≡

```
METHOD SetReduceCounts (ReduceCounts, UnReduceCounts,
                        RHSToRuleArr: ARRAY OF INTEGER);
  BEGIN
  TheReduceCounts ← ReduceCounts;
  TheUnReduceCounts ← UnReduceCounts;
  TheRHSToRuleArr ← RHSToRuleArr;
  END SetReduceCounts;
```

365a ⟨*Consulting synthetic information* 365a⟩≡

```
METHOD ReduceCount: INTEGER;
  BEGIN
  FOR x ∈ TheReduceCounts DO
    RESULT ← RESULT + x;
    END;
  END ReduceCount;

METHOD UnReduceCount: INTEGER;
  BEGIN
  FOR x ∈ TheUnReduceCounts DO
    RESULT ← RESULT + x;
    END;
  END UnReduceCount;
```

The `Walker` class as presented here is a simple and generic extension of a similar (and similarly named) class in the original YAFL compiler [31]. It provides walkthrough services through a parse tree.

365b ⟨`Walker` *class definition* 365b⟩≡

```
CLASS Walker(Class IN BaseNonTerminal);

  REDEFINE METHOD CREATE (Root: BaseNonTerminal);

  METHOD Current: Class;
    POST
      NO_CHANGE;

  METHOD FetchNext: BOOLEAN;
    POST
      (OLD Current = Λ) ⇒ (Current = Λ),
      (Current ≠ Λ) ⇔ RESULT;

END Walker;
```

`Walker` is a parameterized class, that goes through a subtree, and returns successively all the elements of the subtree that are compatible with its parameter class `Class`. For instance, a `Walker(Declaration)` will only return nodes that are compatible with the `Declaration` class, and skip all others.

When creating a `Walker`, one provides a non-terminal which subtree will be walked through.

366 ⟨`Walker` *class implementation* 366⟩≡

```
CLASS Walker(Class IN BaseNonTerminal);

  VAR
    TheStack: Stack(BaseNonTerminal);
    TheCurrent: Class;

  REDEFINE METHOD CREATE (Root: BaseNonTerminal);
    VAR
      p: BaseNonTerminal;
    BEGIN
    BASE;
    TheStack.CREATE;
    p ← Root;
    WHILE (TheCurrent = Λ) ∧ (p ≠ Λ) DO
      WHAT p OF
        IN Class:
          TheCurrent ← TAG;
          END;
       ELSE
        p ← LocalFetchNext (p);
        END;
      END;
    END CREATE;

  METHOD Current: Class;
    BEGIN
    RESULT ← TheCurrent;
    END Current;

  ⟨Fetching a non-terminal 367⟩

END Walker;
```

Fetching a non-terminal is divided into two methods that operate at different levels:

- `LocalFetchNext` returns the next node in the subtree without caring for type restrictions. It manages `TheStack` to remember the paths to visit when plunging in the tree structure
- `FetchNext` calls `LocalFetchNext` until the current non-terminal is compatible with the class attached as restricting parameter to the `Walker`.

367 ⟨*Fetching a non-terminal* 367⟩≡

```
METHOD LocalFetchNext(From: BaseNonTerminal): BaseNonTerminal;

  METHOD Push (Nt: BaseNonTerminal);
    BEGIN
    IF Nt ≠ Λ THEN
      TheStack.Push(Nt);
      END;
    END Push;

  VAR
    n: BaseNonTerminal;

  BEGIN
  n ← From.Next;
  IF (n ≠ Λ) ∧
     (n.Father ≠ Λ) ∧
     n.Father.IsDirectSon(n) THEN
    n ← Λ;
    END;
  IF From.FirstSon ≠ Λ THEN
    Push(n);
    Push(From.Sibling);
    RESULT ← From.FirstSon;
   ELSIF n ≠ Λ THEN
    Push(From.Sibling);
    RESULT ← n;
   ELSE
    RESULT ← From.Sibling;
    IF (RESULT = Λ) ∧ (TheStack.Depth > 0) THEN
      RESULT ← TheStack.Pop;
      END;
    END;
  END LocalFetchNext;
```

368 ⟨*Fetching a non-terminal* 367⟩+≡

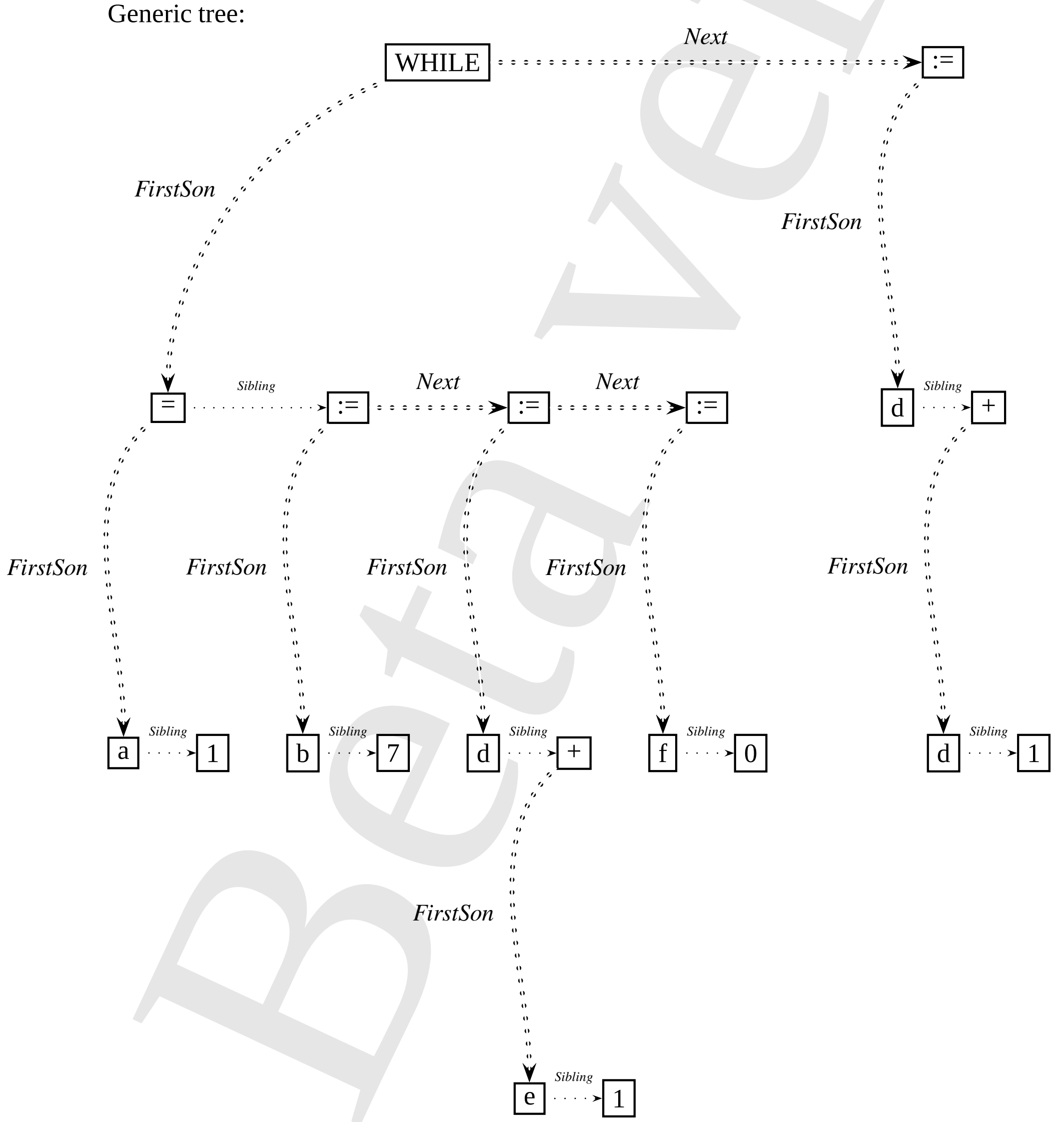
```
METHOD FetchNext: BOOLEAN;
  VAR
    p: BaseNonTerminal;
  BEGIN
  p ← TheCurrent;
  TheCurrent← Λ;
  WHILE (TheCurrent = Λ) ∧ (p ≠ Λ) DO
    p ← LocalFetchNext(p);
    IF p ≠ Λ THEN
      WHAT p OF
        IN Class:
          TheCurrent ← TAG;
          END;
       ELSE
         -- If p is not a Class, go on
         -- searching
        END;
      END;
    END;
  RESULT ← TheCurrent ≠ Λ;
  END FetchNext;
```

Figure 7.3 Generic walkthrough a parse tree

Sample program:

```
WHILE a = 1 DO
  b := 7;
  d := e + 1;
  f := 0;
  END;
d := d + 1;
```

Generic tree:



7.6 DuraCfg

⟨*Definition* ⟩≡

```
DEFINITION MODULE DuraCfg;

FROM YaflNTList IMPORT DeclList;
FROM YaflDeclarations IMPORT ConstDeclaration;
FROM DuraClDefinition IMPORT DuraClassDefinition;
FROM List IMPORT List;

  ONCE CLASS DuraCfg;
    CONST
      DuraExtension = ‘‘cy”;
      DefaultBaseNonTerminalClassName =
        ‘‘DuraBase.BaseNonTerminal”;
      DefaultBaseListClassName       =
        ‘‘DuraBase.NonTerminalList”;
      DefaultLexClassName            =
        ‘‘Lex.Lex”;

    REDEFINE METHOD CREATE;

    ⟨System classes, definition 383c⟩
    ⟨Return the code of a lexeme, given its name, definition 377b⟩
    ⟨Return the smallest code for a lexeme, definition 378c⟩
    ⟨Return the largest code for a lexeme, definition 379b⟩
    ⟨Return the image of a lexeme, given its code, definition 379d⟩
    ⟨Access skeleton names, definition 381a⟩

    METHOD TargetModuleName: ARRAY OF CHAR;
    METHOD TargetClassName: ARRAY OF CHAR;

    METHOD VerboseLevel: INTEGER;
    METHOD ProcessOption (Opt: ARRAY OF CHAR): BOOLEAN;
    METHOD GetClass (Name: ARRAY OF CHAR):
               DuraClassDefinition;

    METHOD NormalizeParser: BOOLEAN;

    ⟨Indicate whether the generated parser is interpreted, definition 382b⟩
    METHOD GenerateVerboseLevel: INTEGER;
    METHOD GenerateDynamicVerbose: BOOLEAN;
    METHOD ConsiderGrammarInfo: BOOLEAN;
    METHOD YaccTargetFName: ARRAY OF CHAR;

  END DuraCfg;
```

END `DuraCfg;`

⟨*Implementation* ⟩≡
```
IMPLEMENTATION MODULE DuraCfg;

FROM Conversions IMPORT IntConversions;
IMPORT Space;
IMPORT ModuleTable;
IMPORT String;
FROM YaflCfg IMPORT YaflCfg, CurrentSpot;
FROM YaflClasses IMPORT ClassDeclaration;
FROM YaflDeclarations IMPORT Declaration;
FROM YaflModules IMPORT DefinitionModule, ImplementationModule;
FROM YaflLiteral IMPORT IntegerLiteral;
FROM Ycc IMPORT CompilationRun;
IMPORT SYSTEM;
FROM YaflSymbols IMPORT SymbolTable;
FROM Streams IMPORT StdOut;

  ONCE CLASS DuraCfg;
    CONST
      DefaultLexName = ‘‘Lex’’;

    ⟨Configuration attributes 373⟩

    METHOD VerboseLevel: INTEGER;
      BEGIN
      RESULT ← YaflCfg.VerboseLevel;
      END VerboseLevel;

    REDEFINE METHOD CREATE;
      BEGIN
      BASE;
      TheNormalizeParserFlag ← TRUE;
      TheLexClassName ← DefaultLexClassName;
      TheBaseListClassName ← DefaultBaseListClassName;
      TheBaseNonTerminalClassName ← DefaultBaseNonTerminalClassName;
      TheConsiderGrammarInfo ← TRUE;
      ⟨Set the default names for the skeleton file names 381c⟩
      END CREATE;

    ⟨System classes, implementation 384⟩

    METHOD NormalizeParser: BOOLEAN;
      BEGIN
      RESULT ← TheNormalizeParserFlag;
      END NormalizeParser;
```

```
    ⟨Indicate whether grammar information must be generated 382a⟩

    ⟨Indicate whether the generated parser is interpreted, implementation 382c⟩
    ⟨Handle the generated parser's verbosity level, implementation 383a⟩

    ⟨Access skeleton names, implementation 381b⟩
    ⟨Target module and class, implementation 386⟩

    ⟨Finding a class using an almost fuzzy scheme 375b⟩
    ⟨Generate a yacc parser, implementation 385b⟩

    ⟨ProcessOption method implementation 374a⟩
    ⟨Return the code of a lexeme, given its name, implementation 378a⟩
    ⟨Return the smallest code for a lexeme, implementation 379a⟩
    ⟨Return the largest code for a lexeme, implementation 379c⟩
    ⟨Return the image of a lexeme, given its code, implementation 380a⟩

  END DuraCfg;

END DuraCfg;
```

373 ⟨*Configuration attributes* 373⟩≡

```
VAR
  TheRootClass,
  TheLexClass,
  TheBaseListClass,
  TheBaseNonTerminalClass: DuraClassDefinition;
  TheParseTableFileName,
  TheRootClassName,
  TheLexClassName,
  TheBaseListClassName,
  TheBaseNonTerminalClassName: ARRAY OF CHAR;
  TheConsiderGrammarInfo,
  TheNormalizeParserFlag: BOOLEAN;
```

374a ⟨ProcessOption *method implementation* 374a⟩≡

```
METHOD ProcessOption (Opt: ARRAY OF CHAR): BOOLEAN;
  VAR
    Words: ARRAY OF ARRAY OF CHAR;
  BEGIN
  -- Default return value is FALSE
  IF Opt.SIZE ≥ 3 THEN
    ASSERT Opt_0 = ''-'';
    IF SYSTEM.UCASE (Opt_1) = ''L'' THEN
      RESULT ← TRUE;
      CASE SYSTEM.UCASE (Opt_2) OF
        ⟨It is a DURA option, check second code 374b⟩
       ELSE
        RESULT ← FALSE;
        END;
      END;
    END;
  END ProcessOption;
```

374b ⟨*It is a* DURA *option, check second code* 374b⟩≡

```
''F'':
  TheParseTableFileName ← Opt.SLICE (3, Opt.SIZE - 3);
  END;
''G'':
  TheConsiderGrammarInfo ← FALSE;
  END;
''N'':
  TheNormalizeParserFlag ← FALSE;
  END;
''L'':
  TheLexClassName ← Opt.SLICE (3, Opt.SIZE - 3);
  END;
''R'':
  TheRootClassName ← Opt.SLICE (3, Opt.SIZE - 3);
  END;
''B'':
  TheBaseNonTerminalClassName ← Opt.SLICE (3, Opt.SIZE - 3);
  END;
''C'':
  TheBaseListClassName ← Opt.SLICE (3, Opt.SIZE - 3);
  END;
```

375a ⟨*It is a* DURA *option, check second code* 374b⟩+≡

```
  ''V'':
    ⟨Handle verbosity level command line option Opt 383b⟩
    END;
  ''S'':
    ⟨Handle skeleton file command line option Opt 381d⟩
    END;
  ''T'':
    ⟨Handle target module and class option 385d⟩
    END;
```

Given a name, GetClass returns a class definition, and ensures that all relevant modules have been analyzed in terms of their semantics.

375b ⟨*Finding a class using an almost fuzzy scheme* 375b⟩≡

```
  METHOD GetClass (Name: ARRAY OF CHAR): DuraClassDefinition;
    VAR
      Module: DefinitionModule;
      Cl: ClassDeclaration;
      MName, CName: ARRAY OF CHAR;
      Words: ARRAY OF ARRAY OF CHAR;
    BEGIN
    IF Name ≠ Λ THEN
      ⟨Name is not void, proceed 376a⟩
      END;
    END GetClass;
```

The module and class names are normalizedusing the string space `Space` as provided by the plain YAFL compiler to allow string comparison to be equivalent to reference comparison. YAFL

376a ⟨`Name` *is not void, proceed* 376a⟩≡
```
Words ← String.BreakInWords (Name,
                                SkipSeparator ← ‘‘.”,
                                KeepSeparator ← Λ);
IF Words.SIZE = 1 THEN
  MName ← Space.StoreString (Name);
  CName ← MName;
 ELSIF Words.SIZE > 1 THEN
  MName ← Space.StoreString (Words0);
  CName ← Space.StoreString (Words1);
  END;
IF MName ≠ Λ THEN
  Module← ModuleTable.LocateDefinitionModule (MName);
  IF Module ≠ Λ THEN
    ⟨Tag and check types for all potentially concerned modules 376b⟩
    Cl ← Module.GetClass (CName);
    IF Cl ≠ Λ THEN
      ⟨Cast Cl into RESULT 377a⟩
      END;
    END;
  END;
```

YAFL This chunk ensures that YAFL semantic and type checking is performed on the transitive closure of the imported modules. This type checking is an idempotent operation: any invocation after the initial invocation has no effect at all.

376b ⟨*Tag and check types for all potentially concerned modules* 376b⟩≡
```
SymbolTable.Check;
Module.TagClosure;
Module.UniqueCheckType;
FOR Mod ∈ Module.TransitiveClosure.Row DO
  Mod.UniqueCheckType;
  END;
```

Since `GetClass` as defined in `DefinitionModule` returns a plain `ClassDefinition`, as opposed to a more specialized `DuraClassDefinition`, we need the following cast, performed by a `WHAT` control statement. This cast if safe, however, since the redefinition of the creators (See module *DuraCreators* on page 404) ensures that whenever `DURA` requires a new `ClassDefinition`, a `DuraClassDefinition` is created.

377a ⟨*Cast* `Cl` *into* `RESULT` 377a⟩≡
```
WHAT Cl OF
  IN DuraClassDefinition:
    RESULT ← TAG;
    END;
  END;
```

377b ⟨*Return the code of a lexeme, given its name, definition* 377b⟩≡
```
METHOD FindLexeme (Name: ARRAY OF CHAR): INTEGER;
```

`FindLexeme` searches the lexemes defined in the class that implements the lexical analyzer, and return the value attached to the constant declaration named `Name`.

378a ⟨*Return the code of a lexeme, given its name, implementation* 378a⟩≡

```
METHOD FindLexeme (Name: ARRAY OF CHAR): INTEGER;
  VAR
    Cl: DuraClassDefinition;
    TheDecl: Declaration;
    a: ARRAY OF CHAR;
  BEGIN
  RESULT ← 0;
  Cl ← LexicalAnalyzerClass;
  CurrentSpot.PushCurrentModule (Cl.Module);
  CurrentSpot.PushCurrentClass (Cl);
  TheDecl ← Cl.GetDecl (Name);
  IF TheDecl ≠ Λ THEN
    WHAT TheDecl OF
      IN ConstDeclaration:
        WHAT TAG.GetFolded OF
          IN IntegerLiteral:
            RESULT ← TAG.Value;
            END;
         ELSE
          END;
        END;
     ELSE
      END;
    END;
  IF RESULT = 0 THEN
    ⟨Consider second chance 378b⟩
    END;
  CurrentSpot.PopCurrentModule;
  END FindLexeme;
```

If no lexeme has been found, it might be due to the fact that the character string that denotes the name has not been normalized, which is required by the YAFL compiler for string matching. By normalizing it on the fly, we also check whether it was normalized before, and if it was not, we perform a recursive call to the same method, hoping it might yield better results this time.

378b ⟨*Consider second chance* 378b⟩≡

```
a ← Space.StoreString (Name);
IF a ≠ Name THEN
  RESULT ← FindLexeme (a);
  END;
```

378c ⟨*Return the smallest code for a lexeme, definition* 378c⟩≡

```
METHOD FirstToken: INTEGER;
```

379a ⟨*Return the smallest code for a lexeme, implementation* 379a⟩≡

```
VAR
  TheFirstToken: INTEGER;

METHOD FirstToken: INTEGER;
  BEGIN
  IF TheFirstToken = 0 THEN
    TheFirstToken ← FindLexeme (Space.StoreString(“FIRSTTOKEN”));
    END;
  RESULT ← TheFirstToken;
  END FirstToken;
```

379b ⟨*Return the largest code for a lexeme, definition* 379b⟩≡

```
METHOD LastToken: INTEGER;
```

379c ⟨*Return the largest code for a lexeme, implementation* 379c⟩≡

```
VAR
  TheLastToken: INTEGER;

METHOD LastToken: INTEGER;
  BEGIN
  IF TheLastToken = 0 THEN
    TheLastToken ← FindLexeme (Space.StoreString(“LASTTOKEN”));
    DEBUG
      IF TheLastToken ≤ 0 THEN
        TheLastToken ← FindLexeme (Space.StoreString(“LASTTOKEN”));
        END;
      END;
    END;
  RESULT ← TheLastToken;
  END LastToken;
```

379d ⟨*Return the image of a lexeme, given its code, definition* 379d⟩≡

```
METHOD LexemeImage (Code: INTEGER): ARRAY OF CHAR;
```

`LexemeImage` is the exact counterpart of `FindLexeme`: given a code, it returns the lexeme's name. It maintains a table `Arr` to cache previously retrieved lexeme names, so that further accesses are instantaneous.

380a ⟨*Return the image of a lexeme, given its code, implementation* 380a⟩≡

```
METHOD LexemeImage (Code: INTEGER): ARRAY OF CHAR;
  VAR
    Cl: DuraClassDefinition;
    TheDecl: Declaration;
    l: DeclList(ConstDeclaration);
    Arr: ONCE ARRAY OF ARRAY OF CHAR;
  BEGIN
  IF Arr = Λ THEN
    Arr.CREATE (LastToken + 1);
    END;
  IF (Code ≥ 0) ∧ (Arr_Code ≠ Λ) THEN
    RESULT ← Arr_Code;
   ELSE
    Cl ← LexicalAnalyzerClass;
    ⟨Search in Cl and its descendants 380b⟩
    IF RESULT = Λ THEN
      RESULT ← IntConversions.IntToString (Code, 0);
      END;
    IF Code ≥ 0 THEN
      Arr_Code ← RESULT;
      END;
    END;
  END LexemeImage;
```

If the lexeme image is not found in `Arr`, `LexemeImage` basically scans the lexer class and its descendants, checks all constant declarations that refer to an integer value, and returns the name of the first of those constants which value matches `Code`.

380b ⟨*Search in* `Cl` *and its descendants* 380b⟩≡

```
WHILE (RESULT = Λ) ∧ (Cl ≠ Λ) DO
  FOR ConstDecl ∈ Cl.Consts  DO
    WHAT ConstDecl.GetFolded OF
      IN IntegerLiteral:
        IF Code = TAG.Value THEN
          RESULT ← ConstDecl.Id.Data;
          END;
        END;
     ELSE
      END;
    END;
  Cl ← Cl.BaseClass;
  END;
```

DURA supports the ability to specify other skeleton names than the default `iskel.dura` and `dskel.dura`. This feature can be used to develop alternate skeletons, tuned for performance, with better reporting facilities, event-driven, multi-threaded, etc.

381a ⟨*Access skeleton names, definition* 381a⟩≡

```
METHOD DefinitionSkeletonName: ARRAY OF CHAR;
METHOD ImplementationSkeletonName: ARRAY OF CHAR;
```

381b ⟨*Access skeleton names, implementation* 381b⟩≡

```
VAR
  TheDefinitionSkeletonName,
  TheImplementationSkeletonName: ARRAY OF CHAR;

METHOD DefinitionSkeletonName: ARRAY OF CHAR;
  BEGIN
  RESULT ← TheDefinitionSkeletonName;
  END DefinitionSkeletonName;

METHOD ImplementationSkeletonName: ARRAY OF CHAR;
  BEGIN
  RESULT ← TheImplementationSkeletonName;
  END ImplementationSkeletonName;
```

381c ⟨*Set the default names for the skeleton file names* 381c⟩≡

```
TheDefinitionSkeletonName ← “dskel.dura”;
TheImplementationSkeletonName ← “iskel.dura”;
```

381d ⟨*Handle skeleton file command line option* Opt 381d⟩≡

```
IF Opt.SIZE ≥ 4 THEN
  CASE SYSTEM.UCASE (Opt3) OF
    “D”:
      TheDefinitionSkeletonName ← Opt.SLICE (4, Opt.SIZE - 4);
      END;
    “I”:
      TheImplementationSkeletonName ← Opt.SLICE (4, Opt.SIZE - 4);
      END;
   ELSE
    RESULT ← FALSE;
    END;
 ELSE
  RESULT ← FALSE;
  END;
```

When compiling using global optimization, grammar information must not be generated. It causes conflicts if the project includes more than one DURA-generated parser, and it does not make much sense anyway.

382a ⟨*Indicate whether grammar information must be generated* 382a⟩≡
```
METHOD ConsiderGrammarInfo: BOOLEAN;
  BEGIN
  RESULT ← TheConsiderGrammarInfo ∧
           ¬ YaflCfg.PleaseGlobalOptimize;
  END ConsiderGrammarInfo;
```

The following two methods can be used to determine whether the user has requested the generation of an interpreted parser (where the push down automaton transitions are stored in an external file) or a compiled parser, where these transitions translate into nested CASE statements, thereby providing better performance but long compilation times as well.

382b ⟨*Indicate whether the generated parser is interpreted, definition* 382b⟩≡
```
METHOD GenerateInterpretedParsers: BOOLEAN;
  POST
    NO_CHANGE,
    RESULT ⇔ (ParseTableFileName ≠ Λ);

METHOD ParseTableFileName: ARRAY OF CHAR;
  POST
    NO_CHANGE;
```

382c ⟨*Indicate whether the generated parser is interpreted, implementation* 382c⟩≡
```
METHOD GenerateInterpretedParsers: BOOLEAN;
  BEGIN
  RESULT ← TheParseTableFileName ≠ Λ;
  END GenerateInterpretedParsers;

METHOD ParseTableFileName: ARRAY OF CHAR;
  BEGIN
  RESULT ← TheParseTableFileName;
  END ParseTableFileName;
```

The verbose level as maintained here is not the verbose level of DURA itself, but rather, the verbose level of the parsers it generates. The verbose level of parsers DURA generates can be dynamic, in the sense that the calling program can tune the verbose level of the parser without regenerating it. It is of course clear that parsers with static verbose level will yield better performance than their dynamic counterparts.

382d ⟨*Configuration attributes* 373⟩+≡
```
TheGenerateVerboseLevel: INTEGER;
```

383a ⟨*Handle the generated parser's verbosity level, implementation* 383a⟩≡

```
METHOD GenerateVerboseLevel: INTEGER;
  BEGIN
  RESULT ← TheGenerateVerboseLevel;
  END GenerateVerboseLevel;

METHOD GenerateDynamicVerbose: BOOLEAN;
  BEGIN
  RESULT ← TheGenerateVerboseLevel < 0;
  END GenerateDynamicVerbose;
```

383b ⟨*Handle verbosity level command line option* `Opt` 383b⟩≡

```
CASE Opt3 OF
  ''d'', ''D'':
    TheGenerateVerboseLevel ← -1;
    END;
  ''0'' .. ''2'':
    TheGenerateVerboseLevel ← SYSTEM.ORD(Opt3) -
                               SYSTEM.ORD(''0'');
    END;
 ELSE
  RESULT ← FALSE;
  END;
```

The following few methods return class definitions (See class *DuraClDefinition. DuraClassDefinition* on page 415) that are used by `DURA` and that must be specified by the user. For instance `RootNonTerminalClass` returns the root class of the grammar, that is, the top level non-terminal that must be returned by a parser.

383c ⟨*System classes, definition* 383c⟩≡

```
METHOD RootNonTerminalClass: DuraClassDefinition;
METHOD BaseNonTerminalClass: DuraClassDefinition;
METHOD BaseListClass: DuraClassDefinition;
METHOD LexicalAnalyzerClass: DuraClassDefinition;
METHOD ResetClasses;
```

These methods use `GetClass` (See page 375) extensively.

384 ⟨*System classes, implementation* 384⟩≡

```
METHOD RootNonTerminalClass: DuraClassDefinition;
  BEGIN
  IF TheRootClass = Λ THEN
    TheRootClass ← GetClass(TheRootClassName);
    END;
  RESULT ← TheRootClass;
  END RootNonTerminalClass;

METHOD BaseNonTerminalClass: DuraClassDefinition;
  BEGIN
  IF TheBaseNonTerminalClass = Λ THEN
    TheBaseNonTerminalClass ←
        GetClass(TheBaseNonTerminalClassName);
    END;
  RESULT ← TheBaseNonTerminalClass;
  END BaseNonTerminalClass;

METHOD BaseListClass: DuraClassDefinition;
  BEGIN
  IF TheBaseListClass = Λ THEN
    TheBaseListClass ← GetClass(TheBaseListClassName);
    ASSERT TheBaseListClass ≠ Λ;
    END;
  RESULT ← TheBaseListClass;
  ASSERT RESULT ≠ Λ;
  END BaseListClass;

METHOD LexicalAnalyzerClass: DuraClassDefinition;
  BEGIN
  IF TheLexClass = Λ THEN
    TheLexClass ← GetClass(TheLexClassName);
    END;
  RESULT ← TheLexClass;
  END LexicalAnalyzerClass;
```

`ResetClasses` removes the references to Y^AF_L objects maintained by `DURA`, in order to let the garbage collector deallocate them is required.

385a ⟨*System classes, implementation* 384⟩+≡

```
METHOD ResetClasses;
  BEGIN
  TheLexClass ← Λ;
  TheRootClass ← Λ;
  TheBaseNonTerminalClass ← Λ;
  TheBaseListClass ← Λ;
  END ResetClasses;
```

385b ⟨*Generate a* `yacc` *parser, implementation* 385b⟩≡

```
VAR
  TheYaccTargetFName: ARRAY OF CHAR;

METHOD YaccTargetFName: ARRAY OF CHAR;
  BEGIN
  RESULT ← TheYaccTargetFName;
  END YaccTargetFName;
```

385c ⟨*It is a* `DURA` *option, check second code* 374b⟩+≡

```
''Y'':
  TheYaccTargetFName ← Opt.SLICE (3, Opt.SIZE - 3);
  END;
```

385d ⟨*Handle target module and class option* 385d⟩≡

```
Words ← String.BreakInWords (Opt.SLICE (3, Opt.SIZE - 3),
                              SkipSeparator ← ''.'',
                              KeepSeparator ← Λ);
IF Words.SIZE = 1 THEN
  TheTargetModuleName ← Words0;
  TheTargetClassName ← Words0;
 ELSIF Words.SIZE = 2 THEN
  TheTargetModuleName ← Words0;
  TheTargetClassName ← Words1;
 ELSE
  RESULT ← FALSE;
  END;
```

`TargetModuleName` and `TargetClassName` return resp. the module name and the class name for the parser `DURA` must generate.

386 ⟨*Target module and class, implementation* 386⟩≡

```
VAR
  TheTargetModuleName,
  TheTargetClassName: ARRAY OF CHAR;

METHOD TargetModuleName: ARRAY OF CHAR;
  BEGIN
  RESULT ← TheTargetModuleName;
  END TargetModuleName;

METHOD TargetClassName: ARRAY OF CHAR;
  BEGIN
  RESULT ← TheTargetClassName;
  END TargetClassName;
```

7.7 DuraCG

YAFL

This module is probably one of the most boring, less interesting and still, most crucial of this entire work. It contains the code generation classes for the grammar extensions. This is not code generation in the common sense, since there is no generation of any executable equivalent here. We use YAFL's externalized code generation mechanisms, to define objects that will be responsible for generating the grammatical information for the various grammatical constructs one stores in a YAFL module.

Code generation as defined here takes these constructs – some of which exist in the plain YAFL compiler, such as `ImplementationModule`, some of which are specific to `DURA`, such as `TerminalSetExpression` – and produce the equivalent intermediate representation (See module *DuraIntermediateFormat* on page 681).

⟨*Definition* ⟩≡
DEFINITION MODULE `DuraCG;`

⟨*Definition import list* 402⟩

⟨`DuraGcContext` *class definition* 388a⟩
⟨`ModuleCodeGenerator` *class definition* 388c⟩
⟨`ClassCodeGenerator` *class definition* 390b⟩
⟨`RuleCodeGenerator` *class definition* 391b⟩
⟨`GrammarExprCodeGenerator` *class definition* 392b⟩
⟨`CaseCodeGenerator` *class definition* 396c⟩
⟨`CaseAltCodeGenerator` *class definition* 397b⟩
⟨`OptionalCodeGenerator` *class definition* 398a⟩
⟨`AggregateCodeGenerator` *class definition* 398c⟩
⟨`TerminalSetCodeGenerator` *class definition* 399b⟩
⟨`RepetitionCodeGenerator` *class definition* 400b⟩
⟨`AtomCodeGenerator` *class definition* 400d⟩
⟨`IdentCodeGenerator` *class definition* 401b⟩

END `DuraCG;`

```
⟨Implementation ⟩≡
  IMPLEMENTATION MODULE DuraCG;

  ⟨Implementation import list 403⟩

  ⟨DuraGcContext class implementation 388b⟩
  ⟨ModuleCodeGenerator class implementation 389a⟩
  ⟨ClassCodeGenerator class implementation 391a⟩
  ⟨RuleCodeGenerator class implementation 392a⟩
  ⟨GrammarExprCodeGenerator class implementation 392c⟩
  ⟨IdentCodeGenerator class implementation 401c⟩
  ⟨CaseCodeGenerator class implementation 397a⟩
  ⟨CaseAltCodeGenerator class implementation 397c⟩
  ⟨AggregateCodeGenerator class implementation 399a⟩
  ⟨TerminalSetCodeGenerator class implementation 400a⟩
  ⟨RepetitionCodeGenerator class implementation 400c⟩
  ⟨OptionalCodeGenerator class implementation 398b⟩
  ⟨AtomCodeGenerator class implementation 401a⟩

  END DuraCG;
```

`DuraGcContext` holds the contextual information which is maintained among all code generators of a pass (in this case, among all code generators for grammatical information applied to a given module). Typically, such a context might hold register allocation information for a native code generator. As the careful reader might notice, there is not much we need in terms of such contextual information right now, but who knows, things might evolve in the future . . .

```
388a  ⟨DuraGcContext class definition 388a⟩≡
        CLASS DuraGcContext;
          INHERITS GcContext (ImplementationModule);
        END DuraGcContext;

388b  ⟨DuraGcContext class implementation 388b⟩≡
        CLASS DuraGcContext;
          INHERITS GcContext (ImplementationModule);
        END DuraGcContext;

388c  ⟨ModuleCodeGenerator class definition 388c⟩≡
        CLASS ModuleCodeGenerator;
          INHERITS ImplModuleCodeGenerator(DuraGcContext);
          REDEFINE METHOD CreateContext: DuraGcContext;
        END ModuleCodeGenerator;
```

ModuleCodeGenerator is the class that takes care of the generation of the intermediate format grammatical information for implementation modules.

389a ⟨ModuleCodeGenerator *class implementation* 389a⟩≡

```
CLASS ModuleCodeGenerator;
  INHERITS ImplModuleCodeGenerator(DuraGcContext);

  REDEFINE METHOD CreateContext: DuraGcContext;
    BEGIN
    RESULT.CREATE;
    END CreateContext;

  REDEFINE METHOD GeneratedLines: INTEGER;
    BEGIN
    RESULT ← 0;
    END GeneratedLines;

  ⟨Code generation for a ModuleCodeGenerator 389b⟩

END ModuleCodeGenerator;
```

Its code generation happens in two stages: it redefines the generic method GenerateCode that calls BuildIntermediate (See the next page). The result is unparsed (See method *DuraIntermediateFormat.InterComponent.UnParse* on page 685) to produce a Compound (See page 801), which is then saved using the Save method (See page 806) onto a file which name uses DuraExtension (See page 370) as extension (by default, "$.cy$").

For all code generators, the Obj method returns the object the code generator applies to (in this specific case, an ImplementationModule.

389b ⟨*Code generation for a* ModuleCodeGenerator 389b⟩≡

```
REDEFINE METHOD GenerateCode;
  BEGIN
  IF DuraCfg.ConsiderGrammarInfo THEN
    BuildIntermediate.UnParse.CastToCompound.Save (
              Obj.GetWorld.BuildTargetFName (Obj.Id.Data,
                                          DuraCfg.DuraExtension));
    END;
  END GenerateCode;
```

Generating the intermediate representation (See class *DuraIntermediateFormat. InterModule* on page 754) for a module is merely a question of going through all its non-terminal classes (See method *DuraClImplementation.DuraClassImplementation.IsNonTerminal* on page 434), and gathering the equivalent intermediate representations (See method *DuraCG.ClassCodeGenerator.BuildIntermediate* on the facing page) into an `InterModule` (See method *DuraIntermediateFormat.InterModule.Append* on page 755).

390a ⟨*Code generation for a* `ModuleCodeGenerator` 389b⟩+≡

```
METHOD BuildIntermediate: InterModule;
  BEGIN
  RESULT.CREATE;
  RESULT.SetModuleName (Obj.Id.Data);
  FOR Cl ∈ Obj.Classes DO
    WHAT Cl OF
      IN DuraClassImplementation:
        IF TAG.IsNonTerminal THEN
          WHAT TAG.Gc OF
            IN ClassCodeGenerator:
              RESULT.Append (TAG.BuildIntermediate);
              END;  -- In
            END;   -- What
          END;   -- If
        END;   -- In
      END;   -- What
    END;  -- For
  END BuildIntermediate;
```

390b ⟨`ClassCodeGenerator` *class definition* 390b⟩≡

```
CLASS ClassCodeGenerator;
  INHERITS ClassImplCodeGenerator(DuraGcContext);
END ClassCodeGenerator;
```

`ClassCodeGenerator` is responsible for the code generation of class implementations. By construction, it will only be used for non-terminal classes. It merely creates an `InterClause` (See page 748), gives it the cut and recovery level defined in the attached class, and gathers the `InterRules` (See page 742) obtained by using `BuildIntermediate` (See page 392) on all its enclosed rules.

For all code generators, the `Obj` method returns the object the code generator applies to (in this specific case, a `DuraClassImplementation`.

391a ⟨`ClassCodeGenerator` *class implementation* 391a⟩≡

```
CLASS ClassCodeGenerator;
  INHERITS ClassImplCodeGenerator(DuraGcContext);

  METHOD BuildIntermediate: InterClause;
    VAR
      TheObj: DuraClassImplementation;
      Gr: GrammarClause;
    BEGIN
    WHAT Obj OF
      IN DuraClassImplementation:
        TheObj ← TAG;
        END;
      END;
    ASSERT TheObj.IsNonTerminal;
    RESULT.CREATE;
    RESULT.SetClass (TheObj.Definition);
    RESULT.SetCutLevel (TheObj.Definition.CutLevel);
    RESULT.SetRecoveryLevel (TheObj.Definition.RecoveryLevel);
    Gr ← TheObj.Grammar;
    IF (Gr ≠ Λ) THEN
      FOR Rule ∈ Gr.Rules DO
        RESULT.Append(Rule.Gc.BuildIntermediate);
        END;
      END;
    END BuildIntermediate;

END ClassCodeGenerator;
```

391b ⟨`RuleCodeGenerator` *class definition* 391b⟩≡

```
CLASS RuleCodeGenerator;
  INHERITS GC(GrammarRule, DuraGcContext);
END RuleCodeGenerator;
```

It is now getting rather systematic, I presume. Generating the intermediate format for a `GrammarRule` (See page 668) is merely a question of creating an `InterRule` (See page 742), giving it the properties (associativity, precedence, etc.) of the `GrammarRule`, and giving it the intermediate expression (See class *DuraIntermediateFormat.InterExpression* on page 703) produced by `BuildIntermediate` (See the facing page).

392a ⟨`RuleCodeGenerator` *class implementation* 392a⟩≡

```
CLASS RuleCodeGenerator;
  INHERITS GC(GrammarRule, DuraGcContext);

  METHOD BuildIntermediate: InterRule;
    BEGIN
    RESULT.CREATE(Λ);
    RESULT.SetAssociativity(Obj.Associativity);
    RESULT.SetRecoveryLevel(Obj.RecoveryLevel);
    RESULT.SetCutLevel(Obj.CutLevel);
    RESULT.SetPrecedence (Obj.Precedence);
    RESULT.SetExpression (Obj.Expr.Gc.BuildIntermediate);
    END BuildIntermediate;

END RuleCodeGenerator;
```

392b ⟨`GrammarExprCodeGenerator` *class definition* 392b⟩≡

```
CLASS GrammarExprCodeGenerator(Class IN GrammarExpression);
  INHERITS GC(Class, DuraGcContext);
END GrammarExprCodeGenerator;
```

After dealing with the structural stuff (module, class, rule), we now consider how to generate the intermediate representation for lower level constructs, namely, grammatical expressions.

`GrammarExprCodeGenerator` serves as base class for the code generators for all grammatical expressions. It contains generic services, namely, handling the target attribute if present.

392c ⟨`GrammarExprCodeGenerator` *class implementation* 392c⟩≡

```
CLASS GrammarExprCodeGenerator(Class IN GrammarExpression);
  INHERITS GC(Class, DuraGcContext);

  DEFERRED METHOD BuildIntermediateValue: InterExpression;

  ⟨BuildIntermediate method implementation 393⟩
  ⟨BuildIntermediateAttribute method implementation 394⟩

END GrammarExprCodeGenerator;
```

Building the intermediate representation for an expression is divided into two phases:

- Only consider the grammatical expression, by calling `BuildIntermediateValue` that must be redefined for the various derivations of `GrammarExprCodeGenerator`
- Consider a target attribute if present, handle assignment to `THIS` (See page 271) as a special case, and call `BuildIntermediateAttribute` (See the next page) to build the intermediate representation of the target attribute otherwise.

393 ⟨`BuildIntermediate` *method implementation* 393⟩≡

```
METHOD BuildIntermediate: InterExpression;
  BEGIN
  RESULT ← BuildIntermediateValue;
  IF Obj.GenericRecursionTarget THEN
    RESULT.SetTarget (InterThis);
   ELSIF Obj.Target ≠ Λ THEN
    RESULT.SetTarget (BuildIntermediateAttribute
                              (Obj.Target.IsAttribute));
    END;
  END BuildIntermediate;
```

`BuildIntermediateAttribute` builds the intermediate representation attached to an attribute `Attr`, in the form of an `InterAttribute` (See page 735) instance. To do so, it must look at `Attr`'s type, and use one of the intermediate predefined types (`InterIntegerType` (See page 687), `InterBooleanType` (See page 688), `InterStringType` (See page 689)) or synthesize an intermediate list (See class *DuraIntermediateFormat.InterListType* on page 696) or non-terminal (See class *DuraIntermediateFormat.InterNonTerminalType* on page 690) type.

394 ⟨BuildIntermediateAttribute *method implementation* 394⟩≡

```
METHOD BuildIntermediateAttribute (Attr: SingleDataItem):
                                             InterAttribute;
   VAR
     TheClass, Cl: ClassDeclaration;
     Encl: DuraClassDefinition;
     NtType: InterNonTerminalType;
     LType: InterListType;
     TheType: Type;
   BEGIN
   ASSERT Attr ≠ Λ;
   ASSERT Attr.Context ≠ Λ;
   RESULT.CREATE;
   ⟨Gather enclosing class definition in Encl 395a⟩
   ⟨First, set the intermediate attribute's name 395b⟩
   TheType ← Attr.GetType;
   IF TheType.ArrayLevel > 0 THEN
     ⟨Check for the ARRAY OF CHAR type 395c⟩
    ELSE
     TheClass ← TheType.SimpleType.ClosestCanonic;
     IF TheClass = PredefItems.Integer THEN
       RESULT.SetType (InterIntegerType);
      ELSIF TheClass = PredefItems.Boolean THEN
       RESULT.SetType (InterBooleanType);
      ELSE
       ⟨Well, it's not a predefined type nor an array 395d⟩
       END;
     END;
   END BuildIntermediateAttribute;
```

The context of `Attr` is a class implementation, but by having replaced the parser for `ClassImplementation` by a `DURA`-specific version (See method *DuraLookAhead.DuraStreamLookAhead.AcceptClassImplementation* on page 816) that creates and returns a `DuraClassImplementation` whenever a plain `ClassImplementation` is required, the following `WHAT` is typewise safe. (See page 434)

395a ⟨*Gather enclosing class definition in* `Encl` 395a⟩≡
```
WHAT Attr.Context OF
  IN DuraClassImplementation:
    Encl ← TAG.Definition;
    END;
  END;
```

395b ⟨*First, set the intermediate attribute's name* 395b⟩≡
```
RESULT.SetNames (Attr.Id.Data,
                 Encl.Id.Data,
                 Encl.Module.Id.Data);
```

395c ⟨*Check for the* `ARRAY OF CHAR` *type* 395c⟩≡
```
ASSERT TheType.Match(PredefItems.ArrayOfChar);
RESULT.SetType (InterStringType);
```

When the target's type is not a predefined type, nor an array, there are still two possibilities:

- It may refer to a `BaseNonTerminal`-derived (See page 346) instance
- It may refer to a `NonTerminalList`-derived (See page 360) instance, to represent repetitions using the "*" or "+" operator.

In any case, it must be public, though...

395d ⟨*Well, it's not a predefined type nor an array* 395d⟩≡
```
ASSERT TheClass.Public;
```

Let's simply check whether `TheClass` derives from the base class for all lists, as identified by the `DuraCfg` module.

395e ⟨*Well, it's not a predefined type nor an array* 395d⟩+≡
```
ASSERT DuraCfg.BaseListClass ≠ Λ;
IF DuraCfg.BaseListClass.Compatible(TheClass) THEN
   ⟨Congratulation ! It's a List !!! 396b⟩
  ELSE
   ⟨Let's ensure it is a non-terminal ... 396a⟩
  END;
```

`ClosestCanonic` is a method that returns the attached class definition when applied to a public class implementation, and the class declaration it is applied to in all other cases. Since non-terminal classes are always public, it is safe to use a `WHAT` to constrain its result to a `DuraClassDefinition`.

396a ⟨*Let's ensure it is a non-terminal . . .* 396a⟩≡
```
ASSERT DuraCfg.BaseNonTerminalClass.Compatible(TheClass);
NtType.CREATE;
WHAT TheClass.ClosestCanonic OF
  IN DuraClassDefinition:
    NtType.SetClass (TAG);
    END;
  END;
RESULT.SetType (NtType);
```

396b ⟨*Congratulation ! It's a* `List` *!!!* 396b⟩≡
```
LType.CREATE;
WHAT TheClass OF
  IN DuraClassDefinition:
    LType.SetTrueClass (TAG);
    END;
  END;
IF TheType.Actuals ≠ Λ THEN
  FOR Class ∈ TheType.Actuals.ActualList DO
    Cl ← Class.Class.Canonic;
    ASSERT Cl.Public;
    WHAT Cl OF
      IN DuraClassDefinition:
        LType.AppendActual (TAG);
        END;
      END;
    END;
  END;
RESULT.SetType (LType);
```

396c ⟨`CaseCodeGenerator` *class definition* 396c⟩≡
```
CLASS CaseCodeGenerator;
  INHERITS GrammarExprCodeGenerator(CaseExpression);
END CaseCodeGenerator;
```

CaseCodeGenerator is responsible for generating the intermediate representation for the grammatical CASE expression. BuildIntermediateValue, as redefined by CaseCodeGenerator, builds an InterCaseExpression (See page 723) instance, gives it the intermediate representation deduced from its discriminant attribute, and collects the intermediate representations built out of each of its alternatives (See method *DuraCG.CaseAltCodeGenerator.BuildIntermediate* on the current page).

397a ⟨CaseCodeGenerator *class implementation* 397a⟩≡

```
CLASS CaseCodeGenerator;
  INHERITS GrammarExprCodeGenerator(CaseExpression);

REDEFINE METHOD BuildIntermediateValue: InterCaseExpression;
  BEGIN
  RESULT.CREATE;
  RESULT.SetDiscriminantAttribute (
            BuildIntermediateAttribute(Obj.DiscriminantAttribute));
  FOR Alt ∈ Obj.Alts DO
    RESULT.AppendAlt (Alt.Gc.BuildIntermediate);
    END;
  END BuildIntermediateValue;

END CaseCodeGenerator;
```

397b ⟨CaseAltCodeGenerator *class definition* 397b⟩≡

```
CLASS CaseAltCodeGenerator;
  INHERITS GC(CaseExpressionAlt, DuraGcContext);

END CaseAltCodeGenerator;
```

Isn't this getting awfully boring ? BuildIntermediate builds an InterCaseAlt instance (See page 726) and initializes it with information extracted from the attached CaseExpressionAlt (See page 462).

397c ⟨CaseAltCodeGenerator *class implementation* 397c⟩≡

```
CLASS CaseAltCodeGenerator;
  INHERITS GC(CaseExpressionAlt, DuraGcContext);

  METHOD BuildIntermediate: InterCaseAlt;
    BEGIN
    RESULT.CREATE;
    RESULT.SetExpression (Obj.Expression.Gc.BuildIntermediate);
    RESULT.SetDiscriminantValue (Obj.DiscriminantValue);
    END BuildIntermediate;

END CaseAltCodeGenerator;
```

398a ⟨OptionalCodeGenerator *class definition* 398a⟩≡
```
CLASS OptionalCodeGenerator;
  INHERITS GrammarExprCodeGenerator(OptionalExpression);
END OptionalCodeGenerator;
```

`OptionalCodeGenerator` is the code generator for `OptionalExpression` (See page 451) instances. `BuildIntermediateValue`, as redefined here, returns an `InterOptionalExpression` (See page 718) that has been initialized to reflect the attached `OptionalExpression`.

398b ⟨OptionalCodeGenerator *class implementation* 398b⟩≡
```
CLASS OptionalCodeGenerator;
  INHERITS GrammarExprCodeGenerator(OptionalExpression);

  REDEFINE METHOD BuildIntermediateValue:
                                InterOptionalExpression;
    BEGIN
    RESULT.CREATE;
    RESULT.SetOperand (Obj.Operand.Gc.BuildIntermediate);
    END BuildIntermediateValue;

END OptionalCodeGenerator;
```

398c ⟨AggregateCodeGenerator *class definition* 398c⟩≡
```
CLASS AggregateCodeGenerator;
  INHERITS GrammarExprCodeGenerator(AggregateExpression);
END AggregateCodeGenerator;
```

`AggregateCodeGenerator` is the code generator for `AggregateExpression` (See page 449) instances. `BuildIntermediateValue`, as redefined here, returns an `InterAggregateExpression` (See page 706) that has been initialized to reflect the attached `AggregateExpression`. `BuildIntermediate` (See page 393) is used to build the intermediate representation for the arguments of the aggregate expression.

399a ⟨`AggregateCodeGenerator` *class implementation* 399a⟩≡

```
CLASS AggregateCodeGenerator;
  INHERITS GrammarExprCodeGenerator(AggregateExpression);

  REDEFINE METHOD BuildIntermediateValue:
                     InterAggregateExpression;
    BEGIN
    RESULT.CREATE;
    IF Obj.Operator = LexicalAnalyzer.Or THEN
      RESULT.SetOp (RESULT.DisjunctionOp);
     ELSE
      RESULT.SetOp (RESULT.ConcatOp);
      END;
    FOR i ∈ 0 .. Obj.Args.Size - 1 DO
      RESULT.Append (Obj.Args_i.
                     Gc.BuildIntermediate);
      END;
    END BuildIntermediateValue;

END AggregateCodeGenerator;
```

Well, I guess the general scheme is now clear ... `TerminalSetCodeGenerator` works just like all other code generators defined in this module. `BuildIntermediateValue` is redefined to return an `InterTerminalSetExpression` (See page 712) initialized according to the attached `TerminalSetExpression` (See page 464).

399b ⟨`TerminalSetCodeGenerator` *class definition* 399b⟩≡

```
CLASS TerminalSetCodeGenerator;
  INHERITS GrammarExprCodeGenerator(TerminalSetExpression);
END TerminalSetCodeGenerator;
```

400a ⟨TerminalSetCodeGenerator *class implementation* 400a⟩≡
```
CLASS TerminalSetCodeGenerator;
  INHERITS GrammarExprCodeGenerator(TerminalSetExpression);

  REDEFINE METHOD BuildIntermediateValue: InterTerminalSetExpression;
    BEGIN
    RESULT.CREATE;
    RESULT.SetTerminalSet (Obj.GetTerminalSet);
    END BuildIntermediateValue;

END TerminalSetCodeGenerator;
```

RepetitionCodeGenerator works just like all other code generators defined in this module. BuildIntermediateValue is redefined to return an InterRepetitionExpression (See page 720) initialized according to the attached RepetitionExpression (See page 453).

400b ⟨RepetitionCodeGenerator *class definition* 400b⟩≡
```
CLASS RepetitionCodeGenerator;
  INHERITS GrammarExprCodeGenerator(RepetitionExpression);
END RepetitionCodeGenerator;
```

400c ⟨RepetitionCodeGenerator *class implementation* 400c⟩≡
```
CLASS RepetitionCodeGenerator;
  INHERITS GrammarExprCodeGenerator(RepetitionExpression);

  REDEFINE METHOD BuildIntermediateValue:
                      InterRepetitionExpression;
    BEGIN
    RESULT.CREATE;
    RESULT.SetAtLeastOnce (Obj.AtLeastOnce);
    RESULT.SetOperand (Obj.Operand.Gc.BuildIntermediate);
    IF Obj.Separator ≠ Λ THEN
      RESULT.SetSeparator (Obj.Separator.
                               Gc.BuildIntermediate);
      END;
    END BuildIntermediateValue;

END RepetitionCodeGenerator;
```

400d ⟨AtomCodeGenerator *class definition* 400d⟩≡
```
CLASS AtomCodeGenerator;
  INHERITS GrammarExprCodeGenerator(AtomExpression);
END AtomCodeGenerator;
```

`AtomCodeGenerator` is responsible for generating code for `AtomExpression` instances (See page 456). It basically takes two cases into account:

- If the `AtomExpression` refers to a class, `BuildIntermediateValye` returns an `InterNonTerminalExpression` (See page 715).
- Otherwise, assuming the `AtomExpression` refers to a lexeme, `BuildIntermediateValue` returns an `InterTerminalExpression` (See page 710).

401a ⟨`AtomCodeGenerator` *class implementation* 401a⟩≡

```
CLASS AtomCodeGenerator;
  INHERITS GrammarExprCodeGenerator(AtomExpression);

  REDEFINE METHOD BuildIntermediateValue: InterExpression;
    VAR
      TheNonTerminalExpr: InterNonTerminalExpression;
      TheTerminalExpr:    InterTerminalExpression;
    BEGIN
    IF Obj.IsClass ≠ Λ THEN
      ⟨Producing an InterNonTerminalExpression 401d⟩
     ELSE
      ⟨Producing an InterTerminalExpression 401e⟩
      END;
    END BuildIntermediateValue;

END AtomCodeGenerator;
```

Well, I'm not really sure what this class actually does. Not much I'm afraid, but removing it would cause quite a few additional problems that are not worth the trouble . . .

401b ⟨`IdentCodeGenerator` *class definition* 401b⟩≡

```
CLASS IdentCodeGenerator;
  INHERITS GC(DuraIdent, DuraGcContext);
END IdentCodeGenerator;
```

401c ⟨`IdentCodeGenerator` *class implementation* 401c⟩≡

```
CLASS IdentCodeGenerator;
  INHERITS GC(DuraIdent, DuraGcContext);
END IdentCodeGenerator;
```

401d ⟨*Producing an* `InterNonTerminalExpression` 401d⟩≡

```
TheNonTerminalExpr.CREATE;
TheNonTerminalExpr.SetClass(Obj.IsClass);
RESULT ← TheNonTerminalExpr;
```

401e ⟨*Producing an* `InterTerminalExpression` 401e⟩≡

```
ASSERT Obj.IsLexeme > 0;
TheTerminalExpr.CREATE;
TheTerminalExpr.Set(Obj.IsLexeme, Obj.Id.FirstIdent.Data);
RESULT ← TheTerminalExpr;
```

402 ⟨*Definition import list* 402⟩≡

```
FROM DuraIdentifiers IMPORT DuraIdent;
FROM DuraExpressions IMPORT CaseExpression, CaseExpressionAlt,
                            AggregateExpression,
                            AtomExpression, OptionalExpression,
                            TerminalSetExpression,
                            RepetitionExpression,
                            GrammarExpression;
FROM DuraGrClause IMPORT GrammarRule;

FROM YaflClImplementation IMPORT ClassImplementation;
FROM YaflCreator IMPORT CodeGeneratorCreator;
FROM YaflModules IMPORT CompilationUnit,
                        ImplementationModule,
                        DefinitionModule;
FROM YaflGC    IMPORT YaflGC;
FROM YaflGCode IMPORT GC, GcContext;
FROM YaflGModules IMPORT ImplModuleCodeGenerator;
FROM YaflGClass IMPORT ClassImplCodeGenerator;
```

403 ⟨*Implementation import list* 403⟩≡

```
IMPORT DuraCfg;
FROM DuraClDefinition     IMPORT DuraClassDefinition;
FROM DuraClImplementation IMPORT DuraClassImplementation;
FROM DuraExpressions      IMPORT GrammarExpression;
FROM DuraGrClause         IMPORT GrammarClause;
FROM DuraIntermediateFormat  IMPORT InterAttribute, InterThis,
         InterTerminalExpression, InterTerminalSetExpression,
         InterNonTerminalExpression,
         InterAggregateExpression,      InterCaseExpression,
          InterCaseAlt, InterRepetitionExpression,
         InterOptionalExpression, InterNonTerminalType,
         InterStringType, InterIntegerType, InterBooleanType,
          InterExpression, InterRule, InterClause,
         InterModule, InterListType;

FROM Streams                IMPORT StdOut, Stream;
FROM YaflModules            IMPORT CompilationUnit,
         ImplementationModule, DefinitionModule, ModuleSet;
FROM YaflCfg                IMPORT CurrentSpot, YaflCfg;
FROM YaflCreator            IMPORT Creators;
FROM YaflClImplementation   IMPORT ClassImplementation;
FROM YaflClDefinition       IMPORT ClassDefinition;
FROM YaflClasses            IMPORT ClassDeclaration;
FROM YaflDeclarations       IMPORT SingleDataItem;
FROM YaflLex                IMPORT LexicalAnalyzer;
FROM YaflNTList             IMPORT DeclList, MultiDeclList,
                                   ClassList, NTSet;
FROM YaflParser             IMPORT NonTerminal;
FROM YaflPredefined         IMPORT PredefItems;
FROM YaflMethods            IMPORT MethodDeclaration;
FROM YaflParser             IMPORT NonTerminal, Walker;
FROM YaflStatements         IMPORT Statement;
FROM YaflType               IMPORT Type;
FROM String                 IMPORT String;
FROM Conversions            IMPORT IntConversions;

FROM YaflCfg IMPORT YaflCfg;
FROM YaflController IMPORT Controller;

IMPORT SYSTEM;
```

7.8 DuraCreators

YAFL

`DuraCreators` implements *creators* (also known as *factories*, according the terminology defined by Gamma et. al. [84]). These creators are responsible for creating new instances of various classes that must be used in relationship with the plain YAFL compiler.

⟨*Definition* ⟩≡

DEFINITION MODULE `DuraCreators;`

⟨`DuraCreators` *definition import list* 414a⟩

⟨`DuraObjectCreator` *class definition* 405a⟩
⟨`DuraStreamCreator` *class definition* 405c⟩
⟨`DuraCodeGeneratorCreator` *class definition* 406b⟩
⟨`DuraPragmaCreator` *class definition* 411a⟩

END `DuraCreators;`

The YAFL compiler supports four different kinds of creators, which all have a DURA-specific version:

- An `ObjectCreator`-derived class, which is responsible for the creation of the non-terminals of the YAFL parse tree. By using a specific `ObjectCreator`, one can have the YAFL compiler `CREATE` a specialized `ClassDefinition`, such as `DuraClassDefinition` (See page 415) whenever a `ClassDefinition` is required.
- A `StreamLookCreator`-derived class, which is responsible for the creation of lexical analyzers. `DuraStreamCreator` (See the next page) will `CREATE` a `DuraStreamLookAhead` (See page 809) whenever a lexical analyzer will be required.
- One (or more) `CodeGenerator`-derived classes which are responsible for the creation of code-generator that must be then attached to non-terminals (See module *DuraCG* on page 387).
- A `PragmaCreator`-derived class, which is responsible for creating the ad hoc objects that represent pragmas that are found in the code. By using a specific `PragmaCreator`, one can have the YAFL compiler recognize new pragmas.

`YDura` is responsible for asking the YAFL compiler to use these creators instead of its original ones (See method *YDura.Dura.CREATE* on page 290).

⟨*Implementation* ⟩≡
```
IMPLEMENTATION MODULE DuraCreators;

⟨DuraCreators implementation import list 414b⟩

⟨DuraObjectCreator class implementation 405b⟩
⟨DuraStreamCreator class implementation 406a⟩
⟨DuraCodeGeneratorCreator class implementation 407a⟩
⟨DuraPragmaCreator class implementation 411b⟩

END DuraCreators;
```

`DuraObjectCreator` is responsible for ensuring that whenever a `DefinitionModule` is required, a `DuraDefinitionModule` (See page 817) is created instead.

405a ⟨`DuraObjectCreator` *class definition* 405a⟩≡
```
  ONCE CLASS DuraObjectCreator;
    INHERITS ObjectCreator;

    REDEFINE METHOD CreateDefinitionModule: DuraDefinitionModule;
  END DuraObjectCreator;
```

The resulting implementation is rather trivial . . .

405b ⟨`DuraObjectCreator` *class implementation* 405b⟩≡
```
ONCE CLASS DuraObjectCreator;
  INHERITS ObjectCreator;

  REDEFINE METHOD CreateDefinitionModule: DuraDefinitionModule;
    BEGIN
    RESULT.CREATE;
    END CreateDefinitionModule;

END DuraObjectCreator;
```

405c ⟨`DuraStreamCreator` *class definition* 405c⟩≡
```
ONCE CLASS DuraStreamCreator;
  INHERITS StreamCreator;

  REDEFINE METHOD CreateStreamLookAhead
                    (Input: InputStream;
                     KeepComments: BOOLEAN):
                       DuraStreamLookAhead;
END DuraStreamCreator;
```

Similarly, whenever a `StreamLookAhead` is required, having set `DuraStreamCreator` as the current `StreamCreator` ensures that a `DuraStreamLookAhead` (See page 809) is created instead. The implementations is definitely free for any superfluous fantasy.

406a ⟨`DuraStreamCreator` *class implementation* 406a⟩≡

```
ONCE CLASS DuraStreamCreator;
  INHERITS StreamCreator;

  REDEFINE METHOD CreateStreamLookAhead
                    (Input: InputStream;
                     KeepComments: BOOLEAN):
                       DuraStreamLookAhead;
    BEGIN
    RESULT.CREATE (Input, KeepComments);
    END CreateStreamLookAhead;

END DuraStreamCreator;
```

A YAFL system can have several simultaneously active `CodeGeneratorCreators`. When generating code, the YAFL compiler will successively use each of the so-defined code generators in turn.

`DuraCodeGeneratorCreator` is responsible for creating the code generators for the production of intermediate grammatical information. It does not replace the genuine `C` code generator, it comes in addition to it. In other words, besides producing `C` code using the code generator creator which is available in the YAFL compiler, `DURA` will also use `DuraCodeGeneratorCreator` to produce objects which will generate the grammatical information.

406b ⟨`DuraCodeGeneratorCreator` *class definition* 406b⟩≡

```
ONCE CLASS DuraCodeGeneratorCreator;
  INHERITS CodeGeneratorCreator;

  REDEFINE METHOD CreateImplModuleCG
                    (NT: ImplementationModule):
                      ModuleCodeGenerator;
  REDEFINE METHOD CreateClassImplCG
                    (NT: ClassImplementation):
                      ClassCodeGenerator;
END DuraCodeGeneratorCreator;
```

407a ⟨DuraCodeGeneratorCreator *class implementation* 407a⟩≡
```
ONCE CLASS DuraCodeGeneratorCreator;
  INHERITS CodeGeneratorCreator;

  ⟨WhatAmI, as redefined by DuraCodeGeneratorCreator 407b⟩
  ⟨Utility creation methods 407c⟩
  ⟨Create a DURA code generator for a given nonterminal 410a⟩

END DuraCodeGeneratorCreator;
```

YAFL

WhatAmI is a method that is called by the $\mathrm{YAF_L}$ compiler on the code generator creators for debugging purposes.

407b ⟨WhatAmI, *as redefined by* DuraCodeGeneratorCreator 407b⟩≡
```
REDEFINE METHOD WhatAmI: ARRAY OF CHAR;
  BEGIN
  RESULT ← ‘‘Dura code generator creator”;
  END WhatAmI;
```

407c ⟨*Utility creation methods* 407c⟩≡
```
REDEFINE METHOD CreateImplModuleCG
                  (NT: ImplementationModule):
                    ModuleCodeGenerator;
  BEGIN
  RESULT.CREATE(NT);
  END CreateImplModuleCG;

REDEFINE METHOD CreateClassImplCG
                  (NT: ClassImplementation):
                    ClassCodeGenerator;
  BEGIN
  RESULT.CREATE(NT);
  END CreateClassImplCG;

METHOD CreateCaseExpressionGC
                (TheExpr: CaseExpression):
                  CaseCodeGenerator;
  BEGIN
  RESULT.CREATE (TheExpr);
  END CreateCaseExpressionGC;

METHOD CreateCaseExpressionAltGC
                (TheAlt: CaseExpressionAlt):
                  CaseAltCodeGenerator;
  BEGIN
  RESULT.CREATE (TheAlt);
  END CreateCaseExpressionAltGC;
```

408 ⟨*Utility creation methods* 407c⟩+≡

```
METHOD CreateAtomExpressionGC
                  (TheExpr: AtomExpression):
                    AtomCodeGenerator;
  BEGIN
  RESULT.CREATE (TheExpr);
  END CreateAtomExpressionGC;

METHOD CreateOptionalExpressionGC
                  (TheExpr: OptionalExpression):
                    OptionalCodeGenerator;
  BEGIN
  RESULT.CREATE (TheExpr);
  END CreateOptionalExpressionGC;

METHOD CreateRepetitionExpressionGC
                  (TheExpr: RepetitionExpression):
                    RepetitionCodeGenerator;
  BEGIN
  RESULT.CREATE (TheExpr);
  END CreateRepetitionExpressionGC;

METHOD CreateAggregateExpressionGC
                  (TheExpr: AggregateExpression):
                    AggregateCodeGenerator;
  BEGIN
  RESULT.CREATE (TheExpr);
  END CreateAggregateExpressionGC;
```

409 ⟨*Utility creation methods* 407c⟩+≡

```
METHOD CreateTerminalSetExpressionGC
                  (TheExpr: TerminalSetExpression):
                    TerminalSetCodeGenerator;
  BEGIN
  RESULT.CREATE (TheExpr);
  END CreateTerminalSetExpressionGC;

METHOD CreateIdentGC (TheId: DuraIdent):
                        IdentCodeGenerator;
  BEGIN
  RESULT.CREATE (TheId);
  END CreateIdentGC;

METHOD CreateRuleGC (TheRule: GrammarRule):
                       RuleCodeGenerator;
  BEGIN
  RESULT.CREATE (TheRule);
  END CreateRuleGC;
```

DuraCodeGeneratorCreator does not support the creation of a code generator object for *all* non-terminals defined by the plain Y^AF_L compiler, because some of these non-terminals don't produce any grammatical information (think of a WHILE statement, for instance). The following WHAT statement tests the type of the non-terminal NT it receives, and returns a newly created matching code generator.

410a ⟨*Create a* DURA *code generator for a given nonterminal* 410a⟩≡

```
REDEFINE METHOD CreateGC(NT: NonTerminal): GC;
    BEGIN
    WHAT NT OF
      IN GrammarRule:
        RESULT ← CreateRuleGC (TAG);
        END;
      IN CaseExpression:
        RESULT ← CreateCaseExpressionGC (TAG);
        END;
      IN CaseExpressionAlt:
        RESULT ← CreateCaseExpressionAltGC (TAG);
        END;
      IN AggregateExpression:
        RESULT ← CreateAggregateExpressionGC (TAG);
        END;
      IN TerminalSetExpression:
        RESULT ← CreateTerminalSetExpressionGC (TAG);
        END;
      IN OptionalExpression:
        RESULT ← CreateOptionalExpressionGC (TAG);
        END;
      IN RepetitionExpression:
        RESULT ← CreateRepetitionExpressionGC (TAG);
        END;
      IN AtomExpression:
        RESULT ← CreateAtomExpressionGC (TAG);
        END;
      IN DuraIdent:
        RESULT ← CreateIdentGC (TAG);
        END;
     ELSE
      ⟨Use the YAFL creator 410b⟩
      END;
    END CreateGC;
```

If NT is not of any of the DURA-specific classes, the creator for code generators defined by the plain Y^AF_L compiler is invoked.

410b ⟨*Use the Y^AF_L creator* 410b⟩≡

```
RESULT ← BASE(NT);
```

`DuraPragmaCreator` is responsible for creating the object that represent `DURA`-specific pragmas (See page 440).

411a ⟨`DuraPragmaCreator` *class definition* 411a⟩≡
```
CLASS DuraPragmaCreator;
  INHERITS PragmaCreator;

  REDEFINE METHOD CREATE;
END DuraPragmaCreator;
```

411b ⟨`DuraPragmaCreator` *class implementation* 411b⟩≡
```
CLASS DuraPragmaCreator;
  INHERITS PragmaCreator;

  REDEFINE METHOD CREATE;

    BEGIN
    InitKeywords;
    BASE;
    END CREATE;

  ⟨Table handling 411c⟩
  ⟨Actually creating the pragmas 413a⟩

END DuraPragmaCreator;
```

Since we might predict that several additional pragmas will be used in the future, we build a generic and easily extensible construct rather than an ad hoc patch.

This construct is made of a set of constants,

411c ⟨*Table handling* 411c⟩≡
```
CONST
  Snip =  0;
  NoCut = 1;
  Cut = 2;
  AutomaticRecovery = 3;

  MaxKeyword = 6;
```
an array of words . . .

411d ⟨*Table handling* 411c⟩+≡
```
VAR
  Keywords : ARRAY OF ARRAY OF CHAR;
```

and a method that initializes this array

412a ⟨*Table handling* 411c⟩+≡

```
METHOD InitKeywords;

  METHOD Enter (Code: INTEGER;
                Key: ARRAY OF CHAR);
    BEGIN
    Keywords_Code ← Space.StoreString (Key);
    END Enter;

  BEGIN
  Keywords.CREATE (MaxKeyword);
  Enter (Cut, ‘‘Cut”);
  Enter (Snip, ‘‘Snip”);
  Enter (NoCut, ‘‘NoCut”);
  Enter (AutomaticRecovery, ‘‘AutomaticRecovery”);
  END InitKeywords;
```

And another method that searches this array for a given entry, and returns its corresponding code or -1 if the entry cannot be found.

412b ⟨*Table handling* 411c⟩+≡

```
METHOD FindKeyword(Key: ARRAY OF CHAR): INTEGER;
  BEGIN
  ⟨Make sure that Key is in the string space 413b⟩
  RESULT ← FIRST i ∈ 0 .. Keywords.SIZE - 1 • Keywords_i = Key;
  END FindKeyword;
```

Depending on the value of `Keyword`, `CreatePragma` returns a new instance of `CutPragma` (See page 442) (creating variations of the same class depending of the level of cut level which is required), a new instance of `AutomaticRecoveryPragma` (See page 443), or calls `BASE` to consider creating pragmas defined by the plain YAFL compiler.

413a ⟨*Actually creating the pragmas* 413a⟩≡

```
REDEFINE METHOD CreatePragma (Keyword: ARRAY OF CHAR;
                              LineNr, ColNr: INTEGER): Pragma;
  VAR
    Code: INTEGER;
    TheCutPragma: CutPragma;
    TheAutomaticRecovery: AutomaticRecoveryPragma;
  BEGIN
  Code ← FindKeyword (Keyword);
  CASE Code OF
    Cut:
      TheCutPragma.CREATE
               (LineNr, ColNr,
                Level ← DuraClassDefinition.FullCut);
      RESULT ← TheCutPragma;
      END;
    NoCut:
      TheCutPragma.CREATE
               (LineNr, ColNr,
                Level ← DuraClassDefinition.NoCut);
      RESULT ← TheCutPragma;
      END;
    Snip:
      TheCutPragma.CREATE
               (LineNr, ColNr,
                Level ← DuraClassDefinition.SnipCut);
      RESULT ← TheCutPragma;
      END;
    AutomaticRecovery:
      TheAutomaticRecovery.CREATE (LineNr, ColNr);
      RESULT ← TheAutomaticRecovery;
      END;
   ELSE
    RESULT ← BASE (Keyword, LineNr, ColNr);
    END;
  END CreatePragma;
```

`Key` must be stored in the string space in order to allow us to use reference comparison (using "=") to search for the keyword in the table.

413b ⟨*Make sure that* `Key` *is in the string space* 413b⟩≡

```
ASSERT Space.StoreString(Key) = Key;
```

414a ⟨DuraCreators *definition import list* 414a⟩≡

```
FROM DuraCG IMPORT ModuleCodeGenerator, ClassCodeGenerator;
FROM DuraLookAhead IMPORT DuraStreamLookAhead;
FROM DuraModules IMPORT DuraDefinitionModule;
FROM DuraIdentifiers IMPORT DuraIdent;
FROM DuraExpressions IMPORT CaseExpression, CaseExpressionAlt,
                            AggregateExpression,
                            AtomExpression, OptionalExpression,
                            RepetitionExpression,
                                TerminalSetExpression,
                            GrammarExpression;
FROM DuraGrClause IMPORT GrammarRule;
IMPORT DuraPragmas;

FROM YaflClImplementation IMPORT ClassImplementation;
FROM YaflCreator IMPORT ObjectCreator, StreamCreator,
                        CodeGeneratorCreator,
                        PragmaCreator;
FROM YaflModules IMPORT ImplementationModule;

FROM Streams IMPORT InputStream;
```

414b ⟨DuraCreators *implementation import list* 414b⟩≡

```
FROM DuraClDefinition IMPORT DuraClassDefinition;
FROM DuraCG IMPORT ModuleCodeGenerator, ClassCodeGenerator,
                   RuleCodeGenerator,
                   IdentCodeGenerator, CaseCodeGenerator,
                   CaseAltCodeGenerator, TerminalSetCodeGenerator,
                   AggregateCodeGenerator, OptionalCodeGenerator,
                   RepetitionCodeGenerator, AtomCodeGenerator;
FROM DuraPragmas IMPORT CutPragma, AutomaticRecoveryPragma;
IMPORT Space;
FROM YaflGCode IMPORT GC;
FROM YaflParser IMPORT NonTerminal;
FROM YaflPragmas IMPORT Pragma;
```

7.9 DuraClDefinition

```
⟨Definition ⟩≡
  DEFINITION MODULE DuraClDefinition;

  ⟨DuraClDefinition definition import list 433a⟩

    ⟨DuraClassDefinition class definition 415⟩

  END DuraClDefinition;

⟨Implementation ⟩≡
  IMPLEMENTATION MODULE DuraClDefinition;

  ⟨DuraClDefinition implementation import list 433b⟩

    ⟨DuraClassDefinition class implementation 416⟩

  END DuraClDefinition;
```

7.9.1 The basics ...

```
415 ⟨DuraClassDefinition class definition 415⟩≡
  CLASS DuraClassDefinition;
    INHERITS InstClassDefinition;
    METHOD IsNonTerminal: BOOLEAN;

    ⟨Constrained redefinitions, definition 417a⟩

    METHOD BuildIntrinsicFlatRule;
    METHOD BuildInheritedFlatRule;

    METHOD Flat: FlatRule;
    METHOD FlatReference: FlatNonTerminal;
    REDEFINE METHOD CREATE(LineNr, ColNr: INTEGER);

    METHOD InterGrammarClause: InterClause;
    ⟨BaseNonTerminal method definition 420b⟩
    ⟨Semantic analysis for grammars, definition 419b⟩
    ⟨Code generation method definition 428a⟩

    ⟨Cut handling, definition 426a⟩
    ⟨Recovery handling, definition 427a⟩

  END DuraClassDefinition;
```

YAFL

DuraClDefinition inherits from InstClassDefinition as defined by the plain YAFL compiler, and denotes class definitions, that is, the part of the class which is present in definition modules.

The purpose of this class is to actually *replace* InstClassDefinition, that is, whenever the compiler creates an InstClassDefinition, we would like it to create a DuraClassDefinition instead. This replacement mechanism is implemented by the DuraObjectCreator class (See page 405).

DuraClassDefinition provides all the services required to access the grammatical information attached to a class, as well as the related code generation methods.

416 ⟨DuraClassDefinition *class implementation* 416⟩≡

```
CLASS DuraClassDefinition;
  INHERITS InstClassDefinition;

  VAR
  ⟨DuraClassDefinition attributes 418b⟩

  ⟨Parse, as redefined to handle nonterminal classes 418a⟩
  ⟨Semantic analysis for grammars, implementation 420a⟩

  REDEFINE METHOD CREATE(LineNr, ColNr: INTEGER);
    BEGIN
    TheDerivedClassList.CREATE;
    BASE(LineNr, ColNr);
    END CREATE;

  REDEFINE METHOD WhatAmI: ARRAY OF CHAR;
    BEGIN
    RESULT ← ‘‘DuraClassDefinition”;
    END WhatAmI;

  ⟨Managing the attached InterClause 419a⟩

  ⟨Constrained redefinitions, implementation 417b⟩

  ⟨BaseNonTerminal method implementation 421a⟩
  ⟨Creating and returning the flat rule attached to this class 422⟩

  ⟨Code generation method implementation 428b⟩

  ⟨Cut handling, implementation 426c⟩
  ⟨Recovery handling, implementation 427b⟩

END DuraClassDefinition;
```

417a ⟨*Constrained redefinitions, definition* 417a⟩≡
```
REDEFINE METHOD Module: DuraDefinitionModule;
REDEFINE METHOD BaseClass: DuraClassDefinition;
```

These two methods are inherited from `ClassDefinition`, and redefined using *constrained redefinitions*. A redefinition is said to be constrained when the type it returns is a proper subtype of the type returned by the method it redefines. When applied to public methods, constrained redefinitions must be public. Y^A^F_L

Such redefinitions take advantage of the fact that we know that not all `DefinitionModule` in our system are `DuraDefinitionModule`'s (See page 817) and that all `ClassDefinition` are `DuraClassDefinition` (See page 415) thank to the redefinition of the standard YAFL compiler creators (See module *DuraCreators* on page 404).

Redefining these two methods here simplifies the code that use them, by replacing numerous WHAT statements in client code by a single WHAT statement in each of these redefinitions.

417b ⟨*Constrained redefinitions, implementation* 417b⟩≡
```
REDEFINE METHOD Module: DuraDefinitionModule;
  BEGIN
  WHAT BASE OF
    IN DuraDefinitionModule:
      RESULT ← TAG;
      END;
    END;
  END Module;
```

417c ⟨*Constrained redefinitions, implementation* 417b⟩+≡
```
REDEFINE METHOD BaseClass: DuraClassDefinition;
  VAR
    Cl: ClassDeclaration;
  BEGIN
  Cl ← BASE;
  IF Cl ≠ Λ THEN
    WHAT Cl OF
      IN DuraClassDefinition:
        RESULT ← TAG;
        END;
      END;
    END;
  END BaseClass;
```

The only difference between parsing a `DuraClassDefinition` and a plain `ClassDefinition` lies in the recognition of the leading `NONTERMINAL` keyword. This keyword has been added to the YAFL lexer as a user-defined token (See class *DuraLex.DuraLex* on page 794). If it is present, `TheIsNonTerminal` is set to `TRUE`. Then, we proceed with the inherited `Parse` method.

YAFL

Lexemes added to the lexical definition by extensions to the plain YAFL compiler are stored in an ad hoc symbol table managed by the YAFL lexer. When encountering one of these user-defined keywords, the lexer returns the `UdfToken` constant, and the `CurrentUdfCode` must be used to know which of these additional keywords has been recognized.

418a ⟨`Parse`, *as redefined to handle nonterminal classes* 418a⟩≡

```
REDEFINE METHOD Parse (Lkh: LookAhead);
  BEGIN
  IF (Lkh.CurrentToken = LexicalAnalyzer.UdfToken) ∧
     (Lkh.CurrentUdfCode = DuraLex.NonTerminal) THEN
    Lkh.GetToken;
    TheIsNonTerminal ← TRUE;
    END;
  BASE (Lkh);
  END Parse;
```

418b ⟨`DuraClassDefinition` *attributes* 418b⟩≡

```
TheIsNonTerminal: BOOLEAN;
```

`TheIsNonTerminal` can be consulted...

418c ⟨`Parse`, *as redefined to handle nonterminal classes* 418a⟩+≡

```
METHOD IsNonTerminal: BOOLEAN;
  BEGIN
  RESULT ← TheIsNonTerminal;
  END IsNonTerminal;
```

`InterGrammarClause` returns the grammar clause attached to `THIS` by fetching it from the intermediate representation attached to its enclosing module. Please keep in mind that `THIS` is a class definition, not a class implementation ! It holds no grammatical information itself. The best it can do is to retrieve it from the intermediate representation that was saved when compiling the corresponding class implementation.

`InterGrammarClause` works lazily, by storing its result into `TheInterGrammarClause` so that later invocation can return this stored value.

419a ⟨*Managing the attached* `InterClause` 419a⟩≡

```
VAR
  TheInterGrammarClause: InterClause;

METHOD InterGrammarClause: InterClause;
  BEGIN
  IF TheInterGrammarClause = Λ THEN
    TheInterGrammarClause ← Module.IntermediateModule.
                              FindClause(Id.Data);
    IF TheInterGrammarClause ≠ Λ THEN
      TheInterGrammarClause.SetClass (THIS);
      SetCutLevel (TheInterGrammarClause.CutLevel);
      SetRecoveryLevel
                (TheInterGrammarClause.RecoveryLevel);
      END;
    END;
  RESULT ← TheInterGrammarClause;
  END InterGrammarClause;
```

419b ⟨*Semantic analysis for grammars, definition* 419b⟩≡

```
METHOD DuraTag (Spec: SyntacticSpecification);
```

Plain semantic analysis, as implemented by the YAFL compiler, is implemented by a method named `Tag`. `DuraTag` performs semantic analysis for `DURA`-related information only.

It basically performs two tasks:

- It adds THIS to the list of derived classes attached to its base class, so that it can produce the grammar rules that implement reverse inheritance (See 6.6.4).
- It informs YAFL of the current class, and calls DuraTag as defined by InterGrammarClause (See page 754), giving it the SyntacticSpecification (See page 295) it received as parameter.

420a ⟨*Semantic analysis for grammars, implementation* 420a⟩≡

```
VAR
  TheDerivedClassList: ArrayList(DuraClassDefinition);

METHOD DuraTag (Spec: SyntacticSpecification);
VAR
  Cl: DuraClassDefinition;
BEGIN
Cl ← BaseNonTerminal;
IF Cl ≠ Λ THEN
  Cl.TheDerivedClassList.Append (THIS);
  END;
IF InterGrammarClause ≠ Λ THEN
  CurrentSpot.PushCurrentClass (THIS);
  InterGrammarClause.DuraTag (Spec);
  CurrentSpot.PopCurrentClass;
  END;
END DuraTag;
```

420b ⟨BaseNonTerminal *method definition* 420b⟩≡

```
METHOD BaseNonTerminal: DuraClassDefinition;
  POST
    RESULT ≠ Λ ⇒ RESULT.IsNonTerminal;
```

`BaseNonTerminal` returns the closest of `THIS`'s base classes that is marked as being a non-terminal class. In other words, it is the base class when restricted to non-terminal classes only.

421a ⟨`BaseNonTerminal` *method implementation* 421a⟩≡

```
METHOD BaseNonTerminal: DuraClassDefinition;
  VAR
    Cl: DuraClassDefinition;
  BEGIN
  Cl ← THIS.BaseClass;
  WHILE (Cl ≠ Λ) ∧ (RESULT = Λ) DO
    IF Cl.IsNonTerminal THEN
      RESULT ← Cl;
     ELSE
      Cl ← Cl.BaseClass;
      END;
    END;
  END BaseNonTerminal;
```

7.9.2 Managing the corresponding intermediate representation

`TheFlat` holds the flattened representation attached to `THIS` (See class *DuraFlat. FlatRule* on page 480).

421b ⟨`DuraClassDefinition` *attributes* 418b⟩+≡

```
TheFlat: FlatRule;
```

Building this flattened representation happens in two stages. First, the intrinsic rules (understand: the rules defined within the class itself) are taken into account. This operation is performed by `BuildIntrinsicFlatRule` (See below). Then, by reverse inheritance, the grammar rules of all the class's derived classes are pasted into `TheFlat` as well (See page 424).

Each `DuraClassDefinition` maintains a *state* (implemented in `BuildingState`) that indicates where it stands in terms of building its attached flattened representation. This allows us to call the various methods in any order, which is not only convenient but also necessary, because gathering the grammar rules attached to derived classes requires that `BuildInheritedFlatRule` must be called for derived classes before. The state system guarantees `BuildInheritedFlatRule`'s idempotence, thereby considerably simplifying the code. When in doubt, we just call `BuildInheritedFlatRule`. If its has already been called, calling it again will have no effect at all.

422 ⟨*Creating and returning the flat rule attached to this class* 422⟩≡

```
VAR
  BuildingState: INTEGER;

CONST
  NoneBuildingState = 0;
  IntrinsicBuildingState = 971;
  InheritedBuildingState = 972;
```

Building the intrinsing grammar rule happens by producing the flattened representation r_i for all the rules, and building a new rule as:

$$\begin{array}{rcl} Intrinsic & ::= & r_1 \mid \\ & & r_2 \mid \\ & & r_3 \mid \\ & & \ldots \\ & & r_n \end{array}$$

Such a representation is far from being optimal in terms of the number of intermediate non-terminals that are created, but a later pass optimizes the resulting grammar whenever possible, thereby removing the unnecessary synthetic flattened rules.

423a ⟨*Creating and returning the flat rule attached to this class* 422⟩+≡
```
METHOD BuildIntrinsicFlatRule;
  VAR
    FRHS: FlatRHS;
  BEGIN
  IF BuildingState = NoneBuildingState THEN
    BuildingState ← IntrinsicBuildingState;
    ASSERT TheFlat = Λ;
    TheFlat.CREATE;
    TheFlat.SetClass (THIS);
    IF InterGrammarClause ≠ Λ THEN
      FOR Rule ∈ InterGrammarClause.RuleList DO
        FRHS.CREATE;
        FRHS.Append (Rule.FlatReference);
        TheFlat.Append (FRHS);
        END;
     ELSE
      DEBUG
        StdOut.WriteLine (“Intergrammar clause is Void”);
        END;
      END;
    ⟨Attach the snip level and recovery status to all right hand sides 423b⟩
    END;
  END BuildIntrinsicFlatRule;
```

We copy the cut and recovery level for all the right hand sides in the flattened representation.

423b ⟨*Attach the snip level and recovery status to all right hand sides* 423b⟩≡
```
FOR RHS ∈ TheFlat DO
  RHS.SetCutLevel (CutLevel);
  RHS.SetRecoveryLevel (RecoveryLevel);
  END;
```

`BuildInheritedFlatRule` adds the flattened representation of `THIS`'s derived classes to the `THIS`'s flattened representation.

424a ⟨*Creating and returning the flat rule attached to this class* 422⟩+≡

```
METHOD BuildInheritedFlatRule;
  POST
    BuildingState ≥ InheritedBuildingState;
  VAR
    PolyReduct: PolymorphismVoidReduction;
    FRule: FlatRule;
    FRHS: FlatRHS;
  BEGIN
  IF BuildingState < InheritedBuildingState THEN
    BuildIntrinsicFlatRule;
    IF BuildingState = IntrinsicBuildingState THEN
      ⟨First, ensure that all derived classes are up-to-date 424b⟩
      ⟨Attach grammar rules for derived classes 424c⟩
      ⟨Attach the snip level and recovery status to all right hand sides 423b⟩
      BuildingState ← InheritedBuildingState;
      END;
    END;
  END BuildInheritedFlatRule;
```

Before appending the flattened representation of all the derived classes, we must first make sure that these representations are up to date, in the sense that they must include grammar rules pasted from their own derived classes.

424b ⟨*First, ensure that all derived classes are up-to-date* 424b⟩≡

```
FOR Cl ∈ TheDerivedClassList DO
  Cl.BuildInheritedFlatRule;
  END;
```

Then, we copy the rules...

424c ⟨*Attach grammar rules for derived classes* 424c⟩≡

```
FOR DerivedClass ∈ TheDerivedClassList DO
  FRule ← DerivedClass.Flat;
  ASSERT FRule ≠ Λ;
  ⟨Paste the entire right hand sides attached to the derived class 425a⟩
  END;
```

by copying each right hand side of each rule. To each of these newly copied right hand side, we attach a newly created instance of `PolymorphismVoidReduction` (See page 833) that indicates the kind of reduction that must be performed when actually reducing the right hand side.

425a ⟨*Paste the entire right hand sides attached to the derived class* 425a⟩≡

```
FOR RHS ∈ FRule DO
  FRHS ← RHS.CLONE;
  PolyReduct.CREATE (BaseClass ← THIS,
                     DerivedClass);
  FRHS.SetReduction (PolyReduct);
  TheFlat.Append (FRHS);
  END;
```

`Flat` ensures that `TheFlat` has been at least allocated (even if it is not complete yet), and `BuildIntrinsicFlatRule` might still have to be called, and then returns it.

425b ⟨*Creating and returning the flat rule attached to this class* 422⟩+≡

```
METHOD Flat: FlatRule;
  BEGIN
  IF TheFlat = Λ THEN
    BuildIntrinsicFlatRule;
    END;
  RESULT ← TheFlat;
  END Flat;
```

`FlatReference` returns a reference to the result of `Flat` by calling `FlatReference` as defined by `FlatRule` (See page 483). `FlatReference`, as defined here, is used when one wishes to insert a reference to the grammar rule attached to a class definition in another grammar rule. The reference one then inserts is a `FlatNonTerminal` (See page 554).

425c ⟨*Creating and returning the flat rule attached to this class* 422⟩+≡

```
METHOD FlatReference: FlatNonTerminal;
  BEGIN
  RESULT ← Flat.FlatReference;
  END FlatReference;
```

7.9.3 Grammatical properties

The following chunks manage class-level grammatical properties, namely the cut level, and the recovery level. These two levels are implemented in `TheCutLevel` and `TheRecoveryLevel`.

425d ⟨`DuraClassDefinition` *attributes* 418b⟩+≡

```
TheCutLevel,
TheRecoveryLevel: INTEGER;
```

The following constants are just syntactical sugar to access the similarly named constants defined in BaseParser (See page 323).

426a ⟨*Cut handling, definition* 426a⟩≡

```
CONST
  FullCut =    BaseParser.FullCut;
  SnipCut =    BaseParser.SnipCut;
  DefaultCut = BaseParser.DefaultCut;
  NoCut =      BaseParser.NoCut;
```

426b ⟨*Cut handling, definition* 426a⟩+≡

```
METHOD SetCutLevel (CutLevel: INTEGER);
PRE
   (CutLevel = FullCut) ∨
   (CutLevel = DefaultCut) ∨
   (CutLevel = SnipCut) ∨
   (CutLevel = NoCut);

METHOD CutLevel: INTEGER;
POST
  NO_CHANGE;
```

426c ⟨*Cut handling, implementation* 426c⟩≡

```
METHOD SetCutLevel (CutLevel: INTEGER);
  BEGIN
  IF TheCutLevel = 0 THEN
    TheCutLevel ← CutLevel;
    END;
  END SetCutLevel;
```

If CutLevel returns DefaultCut, the cut level attached to its base class as returned by BaseNonTerminal (See page 420) is returned. Unless redefined explicitely, grammar rules inherit from their base class's cut level.

426d ⟨*Cut handling, implementation* 426c⟩+≡

```
METHOD CutLevel: INTEGER;
  BEGIN
  RESULT ← TheCutLevel;
  IF (RESULT = DefaultCut) THEN
    IF BaseNonTerminal ≠ Λ THEN
      RESULT ← BaseNonTerminal.CutLevel;
      IF RESULT ≠ DefaultCut THEN
         SetCutLevel (RESULT);
        END;
      END;
    END;
  END CutLevel;
```

427a ⟨*Recovery handling, definition* 427a⟩≡
```
CONST
  NoRecovery = -1;
  DefaultRecovery = 0;
  AutomaticRecovery = 1;

METHOD SetRecoveryLevel (Level: INTEGER);
METHOD RecoveryLevel: INTEGER;
```

427b ⟨*Recovery handling, implementation* 427b⟩≡
```
METHOD SetRecoveryLevel (Level: INTEGER);
  BEGIN
  ASSERT (Level ≥ NoRecovery) ∧
         (Level ≤ AutomaticRecovery);
  IF TheRecoveryLevel = 0 THEN
    TheRecoveryLevel ← Level;
    END;
  END SetRecoveryLevel;
```

If `RecoveryLevel` returns `DefaultRecovery`, the recovery level attached to its base class as returned by `BaseNonTerminal` (See page 420) is returned. Unless redefined explicitely, grammar rules inherit from their base class's recovery level.

427c ⟨*Recovery handling, implementation* 427b⟩+≡
```
METHOD RecoveryLevel: INTEGER;
  BEGIN
  RESULT ← TheRecoveryLevel;
  IF (RESULT = DefaultRecovery) THEN
    IF BaseNonTerminal ≠ Λ THEN
      RESULT ← BaseNonTerminal.RecoveryLevel;
      IF RESULT ≠ DefaultRecovery THEN
         SetRecoveryLevel (RESULT);
        END;
      END;
    END;
  END RecoveryLevel;
```

7.9.4 Code generation

The following chunks describe the various methods provided by `DuraClassDefinition` to produce the code for the grammar information of a class in a `DURA`-generated parser.

Most of these methods take a `DuraOutputStream` as parameter, which is a specialized `OutputStream` that provides numerous pretty-printing facilities to ease the production of reasonably readable YAFL code (See page 859).

428a ⟨*Code generation method definition* 428a⟩≡
```
METHOD YaflClassName: ARRAY OF CHAR;
METHOD GenerateMetaClassVariable (Output: DuraOutputStream);
METHOD GenerateMetaClassVariableDecl (Output: DuraOutputStream);
METHOD GenerateMetaClassVariableInit (Output: DuraOutputStream);

METHOD GenerateCreationMethodName (Output: DuraOutputStream);
METHOD GenerateCreationMethodBody(Output: DuraOutputStream);
```

The `YaflClassName` method returns the name of a class as found within a YAFL source code. If the class holds the same name as its enclosing module, and if this module only includes that class, importing the module automatically imports the class. A module that includes a single public class with the same name, is called a *canonical module*.

Hence, if we are in such a case, we must generate a plain identifier. Otherwise, the qualified identifier $Module.Class$ must be preferred.

428b ⟨*Code generation method implementation* 428b⟩≡
```
METHOD YaflClassName: ARRAY OF CHAR;
  BEGIN
  IF (Module.Canonical) THEN
    RESULT ← Id.Data;
   ELSE
    RESULT ← DotImage;
    END;
  END YaflClassName;
```

YAFL

`GenerateMetaClassVariable` produce a valid YAFL identifier that uniquely identifies the class denoted by `THIS`, by using the `HashValue` method that returns a unique integer for each non-terminal created by a single compilation run.

428c ⟨*Code generation method implementation* 428b⟩+≡
```
METHOD GenerateMetaClassVariable (Output: DuraOutputStream);
  BEGIN
  Output.WriteString (``M_" + IntConversions.IntToString (HashValue, 0));
  END GenerateMetaClassVariable;
```

while `GenerateMetaClassVariableDecl` produces a variable declaration for a reference to the metaclass to `THIS`:

```
M_761: MetaClass(StatsModule.Statement);
```

if `M_761` is the output of `GenerateMetaClassVariable` for the class definition that denotes the `Statement` class in a module `StatsModule`, that is, if the hash value attached to the class definition for `Statement` is 761..

`GenerateMetaClassVariableDecl` also generates the meta-attribute declarations for all the attributes related to the grammatical expression included in `THIS`.

For instance, if a class `IfStatement` includes three attributes `Condition` (of type `Expression`), `Body` and `ElseBody` (of type `StatementList`), `GenerateMetaClassVariableDecl` will produce something of the form:

```
M_871: MetaClass(IfStatement);
A_72: MetaClass(Expression);      -- For Condition
A_73: MetaClass(StatementList);   -- For Body
A_74: MetaClass(StatementList);   -- For ElseBody
```

where `M_871` will be used to access `IfStatements` generically, that is, using the metaclass mechanism as provided by the YAFL standard library; and `A_72`, `A_73` and `A_74` will be used to access the individual attributes of the `IfStatements`.

429a ⟨*Code generation method implementation* 428b⟩+≡

```
METHOD GenerateMetaClassVariableDecl
            (Output: DuraOutputStream);
  VAR
    l: ArrayList(InterExpression);
  BEGIN
  GenerateMetaClassVariable(Output);
  Output.WriteLine (‘‘: MetaClass(” +
                     YaflClassName + ‘‘);”);
  l.CREATE;
  IF InterGrammarClause = Λ THEN
    StdOut.WriteLine (‘‘No grammar clause for: ” + YaflClassName);
    ASSERT FALSE;
   ELSE
    ⟨Find attributes, and generate 429b⟩
    END;
  END GenerateMetaClassVariableDecl;
```

It calls `GrabSubNodes` (See page 685) to gather the list of the sub-expressions involved in this class's grammar rule, `RelatedAttributes` (See page 705) to extract the list of attributes used in each of these sub-expressions, and `GenerateMetaAttributeDecl` (See page 738) to actually produce the corresponding variable declarations.

429b ⟨*Find attributes, and generate* 429b⟩≡

```
InterGrammarClause.GrabSubNodes (l);
FOR Exp ∈ l DO
  FOR Attr ∈ Exp.RelatedAttributes DO
    Attr.GenerateMetaAttributeDecl (Output);
    END;
  END;
```

`GenerateMetaClassVariableInit` produces a chunk of code such as:

```
WHAT DataDictionary.FindMetaClass("StatsModule",
                                  "Statement") OF
  IN MetaClass(StatsModule.Statement) DO
    M_761 := TAG;
    END;
  END;
```

if `M_761` is the output of `GenerateMetaClassVariable` for the class definition that denotes the `Statement` class in a module `StatsModule`.

`GenerateMetaClassVariableInit` also generates the meta-attribute initializations for all the attributes related to the grammatical expression included in `THIS`. It calls `GrabSubNodes` (See page 685) to gather the list of these attributes, and `GenerateMetaAttributeInit` (See page 739) to actually produce the corresponding variable initializations. These initializations rely on the fact that the metaclass variables have been initialized first.

430 ⟨*Code generation method implementation* 428b⟩+≡

```
METHOD GenerateMetaClassVariableInit (Output: DuraOutputStream);
  VAR
    l: ArrayList(InterExpression);
  BEGIN
  Output.WriteLine ("WHAT DataDictionary.FindMetaClass ('" +
                    Module.Id.Data +
                    "', '" + Id.Data + "') OF");
  Output.IncIndent (2);
  Output.WriteLine ("IN MetaClass(" + YaflClassName + "):");
  Output.IncIndent (2);
  GenerateMetaClassVariable(Output);
  Output.WriteLine (" := TAG;");
  Output.End;
  Output.End;
  IF InterGrammarClause ≠ Λ THEN
    l.CREATE;
    InterGrammarClause.GrabSubNodes (l);
    FOR Expr ∈ l DO
      FOR Attr ∈ Expr.RelatedAttributes DO
        Attr.GenerateMetaAttributeInit(Output, THIS);
        END;
      END;
    END;
  END GenerateMetaClassVariableInit;
```

`GenerateCreationMethodName` produce a valid YAFL identifier that uniquely identifies the class denoted by `THIS`, by using the `HashValue` method that returns a unique integer for each non-terminal created by a single compilation run . . .

431 ⟨*Code generation method implementation* 428b⟩+≡

```
METHOD GenerateCreationMethodName (Output: DuraOutputStream);
  BEGIN
  Output.WriteString (‘‘C_” +
                      IntConversions.IntToString (HashValue, 0));
  END GenerateCreationMethodName;
```

`GenerateCreationMethodBody` produces a chunk of code such as:

```
METHOD C_761: StatsModule.Statement;
  BEGIN
  RESULT.CREATE;
  M_761.SetInstance (RESULT);
  RESULT.SetMetaClass(M_761);
  END C_761;
```

if `M_761` is the output of `GenerateMetaClassVariable`, and `C_761` is the output of `GenerateCreationMethodName` for the class definition that denotes the `Statement` class in a module `StatsModule`.

The method body produced by this method creates a new instance of the class denoted by `THIS`, attaches it to the corresponding meta-class as the current instance, and connects to the corresponding meta-class, by calling the `SetMetaClass` method (See page 348).

432 ⟨*Code generation method implementation* 428b⟩+≡

```
METHOD GenerateCreationMethodBody(Output: DuraOutputStream);
  BEGIN
  Output.WriteString (‘‘METHOD ”);
  GenerateCreationMethodName (Output);
  Output.WriteLine (‘‘: ” + YaflClassName + ‘‘;”);
  Output.IncIndent (2);
  Output.WriteLine (‘‘BEGIN”);

  Output.WriteLine (‘‘RESULT.CREATE;”);

  GenerateMetaClassVariable (Output);
  Output.WriteLine (‘‘.UseInstance(RESULT);”);

  Output.WriteString (‘‘RESULT.SetMetaClass(”);
  GenerateMetaClassVariable (Output);
  Output.WriteLine (‘‘);”);

  Output.WriteString (‘‘END ”);
  GenerateCreationMethodName (Output);
  Output.WriteLine (‘‘;”);
  Output.DecIndent (2);
  Output.WriteLn;
  END GenerateCreationMethodBody;
```

For instance, if the hash value for the YAFL class definition for `WhileStatement` is 551, DURA generates a creation method for `WhileStatement` as follows:

```
METHOD C_551: WhileStatement;
  BEGIN
  RESULT.CREATE;
  M_551.UseInstance (RESULT);
  RESULT.SetMetaClass(RESULT);
  END C_551;
```

Incidentally, this is the place where the code to create all non-terminals is generated. If one wishes to lift the current limitation for non-terminal classes having to support a parameterless CREATE method as explained in 6.3, `GenerateCreationMethod-Body` must be changed.

433a ⟨`DuraClDefinition` *definition import list* 433a⟩≡

```
FROM DuraBase IMPORT BaseParser;
FROM DuraFlat IMPORT FlatRule, FlatNonTerminal;
FROM DuraIntermediateFormat IMPORT InterClause;
FROM DuraOutput IMPORT DuraOutputStream;
FROM DuraModules IMPORT DuraDefinitionModule;
FROM DuraSpecification IMPORT SyntacticSpecification;

FROM Streams IMPORT OutputStream;
FROM YaflClDefinition IMPORT InstClassDefinition;
FROM YaflNTList IMPORT NTList, DeclList, NTSet;
```

433b ⟨`DuraClDefinition` *implementation import list* 433b⟩≡

```
IMPORT ArrayList;
IMPORT DuraCfg;
FROM DuraFlat IMPORT FlatRHS;
FROM DuraIntermediateFormat IMPORT InterModule, InterClause,
                                   InterRule,
                                   InterAttribute, InterExpression,
                                   InterListType, InterThis;
FROM DuraReductions IMPORT PolymorphismVoidReduction;

FROM Conversions IMPORT IntConversions;
FROM LookAhead IMPORT LookAhead;

FROM YaflCfg IMPORT CurrentSpot;
FROM YaflLex IMPORT LexicalAnalyzer;
FROM YaflClasses IMPORT ClassDeclaration;

IMPORT DuraLex;
IMPORT String;
FROM Streams IMPORT StdOut;
```

7.10 DuraClImplementation

This module defines a derivation of the `ClassImplementation` class, as defined by the plain YAFL compiler. For external modules, this derivation, named `DuraClassImplementation`, provides the following services:

- Indication whether a class implementation is a non-terminal class (that is, whether the class is marked with the `NONTERMINAL` keyword)
- Access to the grammar clause embedded in a non-terminal class

The `DuraObjectCreator` class (See page 405) is responsible for creating a `DuraClassImplementation` instead of a plain `ClassImplementation` every time such an object is to be allocated by the YAFL compiler. According to the design pattern [84] terminology, `DuraObjectCreator` is a factory.

⟨*Definition* ⟩≡

```
DEFINITION MODULE DuraClImplementation;

FROM DuraClDefinition IMPORT DuraClassDefinition;
FROM DuraGrClause IMPORT GrammarClause;

FROM YaflClImplementation IMPORT ClassImplementation;

  CLASS DuraClassImplementation;
    INHERITS ClassImplementation;

    METHOD IsNonTerminal: BOOLEAN;
      POST
        NO_CHANGE;

    METHOD Grammar: GrammarClause;
      POST
        ¬ IsNonTerminal ⇒ RESULT = Λ;

    REDEFINE METHOD Definition: DuraClassDefinition;

  END DuraClassImplementation;

END DuraClImplementation;
```

```
⟨Implementation ⟩≡
  IMPLEMENTATION MODULE DuraClImplementation;

  ⟨DuraClImplementation implementation import list 439c⟩

    CLASS DuraClassImplementation;
      INHERITS ClassImplementation;

      VAR
        TheGrammar: GrammarClause;
        TheIsNonTerminal: BOOLEAN;

      METHOD IsNonTerminal: BOOLEAN;
        BEGIN
        RESULT ← TheIsNonTerminal;
        END IsNonTerminal;

      METHOD Grammar: GrammarClause;
        BEGIN
        RESULT ← TheGrammar;
        END Grammar;

      ⟨Parsing, as modified by DuraClImplementation 437b⟩
      ⟨Tagging a DURA class implementation 435⟩
      ⟨Other redefinitions 438c⟩
      ⟨Definition constrained redefinition 439b⟩

    END DuraClassImplementation;

  END DuraClImplementation;
```

YAFL

In the YAFL terminology, *tagging* is semantic analysis except for type checking which is handled by the `CheckType` method (See page 438). Hence, tagging a `DuraClassImplementation` first means calling `BASE` to reuse whatever analysis was performed for plain YAFL class implementations.

435

```
⟨Tagging a DURA class implementation 435⟩≡
  REDEFINE METHOD Tag;
    BEGIN
    BASE;
```

We then ensure that `NONTERMINAL` marking are consistent between definition and implementation for the class . . .

436a ⟨*Tagging a* DURA *class implementation* 435⟩+≡
```
IF TheIsNonTerminal THEN
   ⟨Make sure the class is public and consistent with its definition 436b⟩
  ELSIF (Definition ≠ Λ) ∧ (Definition.IsNonTerminal) THEN
   Error (‘‘NONTERMINAL clauses must be consistent between ” +
        ‘‘definition and implementation”);
   END;
```

436b ⟨*Make sure the class is public and consistent with its definition* 436b⟩≡
```
IF Definition = Λ THEN
   Error (‘‘NONTERMINAL classes must be public”);
  ELSIF ¬ Definition.IsNonTerminal THEN
   Error (‘‘NONTERMINAL clauses must be consistent between ” +
          ‘‘definition and implementation”);
   END;
```

and finally, if a grammar rule is specified, tag it as well.

436c ⟨*Tagging a* DURA *class implementation* 435⟩+≡
```
IF TheGrammar ≠ Λ THEN
   ⟨Tag the grammar rule 436d⟩
   END;
END Tag;
```

Since the grammar rule can use attributes as well as constants (in grammatical `CASE` tags, for instance), before tagging the rule by calling `UniqueTag`, we define a new scope in the symbol table, and enter all declarations from the class implementation, as well as the constants defined in the attached definition module so that all the information the grammar rules can refer to are accessible.

436d ⟨*Tag the grammar rule* 436d⟩≡
```
SymbolTable.PushLevel;
SubDecls.Enter;
IF Definition ≠ Λ THEN
   Definition.Consts.Enter;
   END;
This.Enter;
TheGrammar.UniqueTag;
SymbolTable.PopLevel;
```

To cope with cyclic imports – which are acceptable in YAFL, unlike Modula II or Ada – tagging is split in several methods which are invoked on the transitive closure of the imported modules. `TagInheritsClause` is one of these methods. It is redefined here to ensure that a class marked as `NONTERMINAL` also inherits from the right base class (See method *DuraCfg.DuraCfg.BaseNonTerminalClass* on page 383).

437a ⟨*Tagging a* `DURA` *class implementation* 435⟩+≡

```
REDEFINE METHOD TagInheritsClause;
  VAR
    p: ClassDeclaration;
  BEGIN
  BASE;
  IF TheIsNonTerminal THEN
    p ← BaseClass;
    ASSERT DuraCfg.BaseNonTerminalClass ≠ Λ;
    WHILE (p ≠ Λ) ∧ (p ≠ DuraCfg.BaseNonTerminalClass) DO
      p ← p.BaseClass;
      END;
    IF p = Λ THEN
      Error (“A NONTERMINAL must inherit from BaseNonTerminal”);
      END;
    END;
  END TagInheritsClause;
```

YAFL

The following chunks describe how the grammatical extensions can be implemented, that is, how we can ask the YAFL compiler to recognize extensions to its original syntax.

`Parse` is redefined to detect the `NONTERMINAL` keyword before starting to parse the class, set the appropriate flag accordingly, then call `BASE` to proceed with normal parsing.

437b ⟨*Parsing, as modified by* `DuraClImplementation` 437b⟩≡

```
REDEFINE METHOD Parse (Lkh: LookAhead);
  BEGIN
  IF  (Lkh.CurrentToken = LexicalAnalyzer.UdfToken) ∧
         (Lkh.CurrentUdfCode = DuraLex.NonTerminal) THEN
    Lkh.GetToken;
    TheIsNonTerminal ← TRUE;
    END;
  BASE (Lkh);
  END Parse;
```

YAFL

The original `ParseDeclarationList` method is called after the header of the class, to gather the various declarations (constants, variables, methods) declared in the class. It is redefined here, to accept a *leading* grammar clause, before proceeding with the original behaviour by calling `BASE`.

438a ⟨*Parsing, as modified by* `DuraClImplementation` 437b⟩+≡

```
REDEFINE METHOD ParseDeclarationList
    (Lkh: LookAhead;
     Implementation: BOOLEAN);
  BEGIN
  ASSERT Implementation;
  ⟨If we start with the word GRAMMAR, parse the grammar rule 438b⟩
  BASE (Lkh, Implementation);
  END ParseDeclarationList;
```

YAFL

When redefining a YAFL method, the parameter names and types must be preserved. Hence, `Lkh` is defined as a `LookAhead`, while we know, by having redefined the standard creators (See class *DuraCreators.DuraStreamCreator* on page 405) that all lookahead instances in our system will be `DuraStreamLookAhead` (See page 809) instances. Hence, it is safe to constrain `Lkh` by using a `WHAT` control structure, so that we can take advantage of the extra features provided by `DuraStreamLookAhead` (namely, `AcceptGrammarClause` (See page 810)).

438b ⟨*If we start with the word* GRAMMAR, *parse the grammar rule* 438b⟩≡

```
IF (Lkh.CurrentToken = LexicalAnalyzer.UdfToken) ∧
   (Lkh.CurrentUdfCode = DuraLex.Grammar) THEN
  WHAT Lkh OF
    IN DuraStreamLookAhead:
      TheGrammar ← TAG.AcceptGrammarClause;
      SetSon (TheGrammar);
      IF ¬ IsNonTerminal THEN
        Error(“Grammar clauses can only occur within ” +
               “NONTERMINAL classes”);
        END;
      END;
    END;
  END;
```

YAFL

`CheckType` performs type checking. It is redefined here to check the type of the grammar clause a well, if present.

438c ⟨*Other redefinitions* 438c⟩≡

```
REDEFINE METHOD CheckType;
  BEGIN
  BASE;
  IF TheGrammar ≠ Λ THEN
    TheGrammar.UniqueCheckType;
    END;
  END CheckType;
```

SubTree returns an array of the sub-nodes of every non-terminal in the YAFL parse tree. It is redefined here to include the grammar clause in these sub-nodes if present. YAFL

439a ⟨*Other redefinitions* 438c⟩+≡

```
REDEFINE METHOD SubTree: ARRAY OF NonTerminal;
  BEGIN
  IF TheGrammar = Λ THEN
    RESULT ← BASE;
   ELSE
    RESULT.CREATE (1);
    RESULT0 ← TheGrammar;
    RESULT ← BASE + RESULT;
    END;
  END SubTree;
```

YAFL

Definition is the method that returns the class definition attached to a public class implementation, and VOID when applied to a private class implementation. It is redefined here as a constrained redefinition to return a DuraClassDefinition (See page 415) rather than a plain ClassDefinition. This operation serves mainly a convenience purpose, and is entirely safe since DuraObjectCreator (See page 405) ensures that whenever the YAFL compiler needs a ClassDefinition to be created, a DuraClassDefinition is created instead.

439b ⟨Definition *constrained redefinition* 439b⟩≡

```
REDEFINE METHOD Definition: DuraClassDefinition;
  BEGIN
  IF BASE ≠ Λ THEN
    WHAT BASE OF
      IN DuraClassDefinition:
        RESULT ← TAG;
        END;
      END;
    END;
  END Definition;
```

439c ⟨DuraClImplementation *implementation import list* 439c⟩≡

```
IMPORT DuraCfg;
FROM YaflLex IMPORT LexicalAnalyzer;
IMPORT DuraLex;
FROM DuraLookAhead IMPORT DuraStreamLookAhead;
FROM LookAhead IMPORT LookAhead;
FROM YaflParser IMPORT NonTerminal;
FROM YaflSymbols IMPORT SymbolTable;
IMPORT String;
FROM YaflClasses IMPORT ClassDeclaration;
FROM Streams IMPORT StdOut;
```

7.11 DuraPragmas

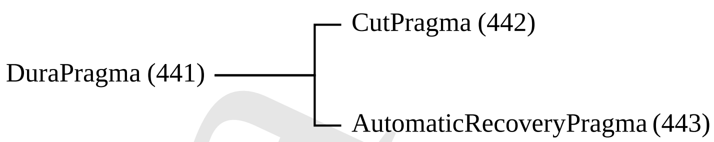
YAFL

Pragmas are named compiler directives for which a normalized syntax is recognized by the YAFL compiler. Creators are used to create `Pragma`'s depending on the pragma name that is specified in the source (See class *DuraCreators.DuraPragmaCreator* on page 411). This module contains the definition for the two DURA-specific pragmas.

⟨*Definition* ⟩≡

```
DEFINITION MODULE DuraPragmas;

FROM YaflPragmas IMPORT Pragma;
FROM DuraBase IMPORT BaseParser;

  ⟨DuraPragma base class definition 441a⟩

    ⟨CutPragma class definition 442c⟩
    ⟨AutomaticRecoveryPragma class definition 443c⟩

END DuraPragmas;
```

Figure 7.4 Inheritance tree for DURA's specific pragmas



⟨*Implementation* ⟩≡

```
IMPLEMENTATION MODULE DuraPragmas;

FROM DuraClDefinition IMPORT DuraClassDefinition;
IMPORT Ref;
FROM YaflClasses IMPORT ClassDeclaration;

⟨DuraPragma base class implementation 441b⟩

  ⟨CutPragma class implementation 443a⟩
  ⟨AutomaticRecoveryPragma class implementation 444a⟩

END DuraPragmas;
```

`DuraPragma` serves as base class for all `DURA`-specific pragmas.

441a ⟨`DuraPragma` *base class definition* 441a⟩≡
```
CLASS DuraPragma;
  INHERITS Pragma;
END DuraPragma;
```

It provides basic services, such as returning the enclosing class definition of the pragmas.

441b ⟨`DuraPragma` *base class implementation* 441b⟩≡
```
CLASS DuraPragma;
  INHERITS Pragma;

  ⟨EnclosingClass method implementation 441c⟩
  ⟨Tag, as redefined by DuraPragma 442b⟩

END DuraPragma;
```

`EnclosingClass` will return the enclosing class definition of the pragma, by first getting its enclosing class *implementation*, then ensuring that this class implementation is public (such pragmas in private classes are invalid anyway), and returning the corresponding definition after ensuring that it was marked as a non-terminal class (See method *DuraClDefinition.DuraClassDefinition.IsNonTerminal* on page 418). If any error has been detected, `EnclosingClass` returns `VOID`.

441c ⟨`EnclosingClass` *method implementation* 441c⟩≡
```
VAR
  TheEnclosingClass: DuraClassDefinition;

METHOD EnclosingClass: DuraClassDefinition;
  VAR
    r: Ref(ClassDeclaration);
  BEGIN
  IF TheEnclosingClass = Λ THEN
    ⟨Get the ancestor in r ... 442a⟩
    IF r.Get ≠ Λ THEN
      IF r.Get.Public THEN
        WHAT r.Get.ClosestCanonic OF
          IN DuraClassDefinition:
            IF TAG.IsNonTerminal THEN
              TheEnclosingClass ← TAG;
               END; -- If
             END; -- In
          END;  -- What
        END;  -- If
      END;  -- If
    END;  -- If
  RESULT ← TheEnclosingClass;
  END EnclosingClass;
```

Getting the enclosing class declaration makes use of a (rather exclusive, as far as I know) feature of YAFL, namely, the ability to dynamically test the argument to a parameterized class for compatibility with an object.

YAFL

The `GetAncestor` method can be applied to any non-terminal in a YAFL parse tree, takes a reference (in the form of an instance of the `Ref` class) and sets the reference to the closest enclosing non-terminal that matches the `Ref`'s class.

For instance, if given a `Ref(Statement)`, it will set it to the closest enclosing `Statement`, if given a `Ref(ClassImplementation)`, it will return the closest enclosing `ClassImplementation`, etc ... The value that has been put into this `Ref` can then be queried using the `Get` method.

442a ⟨*Get the ancestor in r ...* 442a⟩≡

```
r.CREATE(Λ);
GetAncestor(r);
```

In general, pragmas can occur in definition modules as well, but this is not the case for the pragmas listed here. `Tag` is redefined here, to ensure that such an enclosing class exists.

442b ⟨`Tag`, *as redefined by* `DuraPragma` 442b⟩≡

```
REDEFINE METHOD Tag;
  BEGIN
  IF EnclosingClass = Λ THEN
    Error (“DURA pragmas should only be ” +
           “applied within public NONTERMINAL ” +
            “classes”);
    END;
  END Tag;
```

`CutPragma` is the class used to implement the `Cut`, `Snip` and `NoCut` pragmas (See page 271), the difference of behaviour being dictated by the `Level` parameter given at creation time..

442c ⟨`CutPragma` *class definition* 442c⟩≡

```
CLASS CutPragma;
  INHERITS DuraPragma;

  REDEFINE METHOD CREATE(LineNr, ColNr: INTEGER;
                         Level: INTEGER);
    PRE
      (Level = BaseParser.FullCut) ∨
      (Level = BaseParser.SnipCut);

END CutPragma;
```

443a ⟨CutPragma *class implementation* 443a⟩≡

```
CLASS CutPragma;
  INHERITS DuraPragma;

  VAR
    TheLevel: INTEGER;

  REDEFINE METHOD CREATE(LineNr, ColNr: INTEGER;
                         Level: INTEGER);
    BEGIN
    BASE (LineNr, ColNr);
    TheLevel ← Level;
    END CREATE;

  REDEFINE METHOD WhatAmI: ARRAY OF CHAR;
    BEGIN
    RESULT ← ‘‘CutPragma”;
    END WhatAmI;

  ⟨Tagging a CutPragma 443b⟩

END CutPragma;
```

Semantic analysis for a CutPragma basically ensures that no parameter is specified, and transfers the cut level to the enclosing class by means of SetCutLevel (See page 426).

443b ⟨*Tagging a* CutPragma 443b⟩≡

```
REDEFINE METHOD Tag;
  VAR
    ClassDecl: DuraClassDefinition;
  BEGIN
  BASE;
  IF (Param = Λ) ∨ (Param.Size = 0) THEN
    ClassDecl ← EnclosingClass;
    IF ClassDecl ≠ Λ THEN
      ClassDecl.SetCutLevel (TheLevel);
      END;
   ELSE
    Error (‘‘No parameters expected”);
    END;
  END Tag;
```

443c ⟨AutomaticRecoveryPragma *class definition* 443c⟩≡

```
CLASS AutomaticRecoveryPragma;
  INHERITS DuraPragma;

END AutomaticRecoveryPragma;
```

444a ⟨AutomaticRecoveryPragma *class implementation* 444a⟩≡

```
CLASS AutomaticRecoveryPragma;
  INHERITS DuraPragma;

  ⟨Tagging an AutomaticRecoveryPragma 444b⟩

END AutomaticRecoveryPragma;
```

YAFL

It is YAFL's pragmas responsibility to check for the correct number of parameters (by means of the `Params` method, provided by the `Pragma` base class included in the YAFL compiler) they receive. For the rest, this method simply sets its enclosing class (See method *DuraPragmas.DuraPragma.EnclosingClass* on page 441) recovery level.

444b ⟨*Tagging an* AutomaticRecoveryPragma 444b⟩≡

```
REDEFINE METHOD Tag;
  VAR
    ClassDecl: DuraClassDefinition;
  BEGIN
  BASE;
  IF (Param = Λ) ∨ (Param.Size = 0) THEN
    ClassDecl ← EnclosingClass;
    IF ClassDecl ≠ Λ THEN
      ClassDecl.SetRecoveryLevel(ClassDecl.AutomaticRecovery);
      END;
   ELSE
    Error (‘‘No parameters expected”);
    END;
  END Tag;
```

7.12 DuraExpressions

`DuraExpression` includes the classes that describe a grammatical expression at the YAFL compiler level. As such, these grammatical expressions are not suited for parser generation. They must go through several transformations before a parser generation algorithm can be applied.

The classes listed here *extend* the compiler's initial behaviour. Hence, they must integrate gracefully with the YAFL compiler environment, by systematically redefining some methods for semantic analysis, type checking, textual identification (`WhatAmI` method), etc.

```
⟨Definition ⟩≡
  DEFINITION MODULE DuraExpressions;

  ⟨Definition import list 473a⟩

    ⟨GrammarExpression class definition 446⟩
    ⟨AggregateExpression class definition 449⟩
    ⟨TerminalSetExpression class definition 464b⟩
    ⟨OptionalExpression class definition 451b⟩
    ⟨RepetitionExpression class definition 453⟩
    ⟨AtomExpression class definition 456⟩

    ⟨Grammar CASE construct class definitions 459c⟩

  END DuraExpressions;
```

```
⟨Implementation ⟩≡
  IMPLEMENTATION MODULE DuraExpressions;

  ⟨Implementation import list 473b⟩

    ⟨GrammarExpression class implementation 447⟩
    ⟨AggregateExpression class implementation 450⟩
    ⟨OptionalExpression class implementation 452a⟩
    ⟨RepetitionExpression class implementation 454⟩
    ⟨AtomExpression class implementation 457⟩

    ⟨Grammar CASE construct class implementations 460a⟩
    ⟨TerminalSetExpression class implementation 465a⟩
    ⟨Terminal expression intermediate classes 468a⟩

  END DuraExpressions;
```

Figure 7.5 Inheritance tree for DURA's grammatical expressions

GrammarExpression (446)
- AggregateExpression (449)
- OptionalExpression (451)
- RepetitionExpression (453)
- AtomExpression (456)
- CaseExpression (459)
- TerminalSetExpression (464)

`GrammarExpression` is the base class for all grammar expressions that can be built using grammatical operators such as "*", "?", etc... The only generic service it provides is the ability to be attached to a target, as an identifier that must refer to a non-once attribute.

This attachment happens through an identifier (See class *DuraIdentifiers.DuraIdent* on page 673) rather than a variable declaration directly, because this attachment happens at parse time, when no semantic information is available to find the attribute. Deducing the attribute from the identifier happens during semantic analysis, as redefined by `GrammarExpression` (See page 448).

446 ⟨`GrammarExpression` *class definition* 446⟩≡

```
CLASS GrammarExpression(gc IN GrammarExprCodeGenerator);
  INHERITS NonTerminal(gc);

  METHOD SetTarget(Id: DuraIdent);
  METHOD Target: DuraIdent;
  METHOD GenericRecursionTarget: BOOLEAN;
END GrammarExpression;
```

`TheGenericRecursionTarget` is set to `TRUE` by semantic analysis (See method *DuraExpressions.GrammarExpression.Tag* on the following page) when the target is `THIS`, indicating that the grammar rule is of the form:

```
NONTERMINAL CLASS Expression;

GRAMMAR
  LParent THIS := Expression RParent;
...
```

that requires special treatment.

447 ⟨`GrammarExpression` *class implementation* 447⟩≡

```
  CLASS GrammarExpression(gc IN GrammarExprCodeGenerator);
    INHERITS NonTerminal(gc);

    VAR
      TheTarget: DuraIdent;
      TheGenericRecursionTarget: BOOLEAN;

    METHOD SetTarget(Id: DuraIdent);
      BEGIN
      TheTarget ← Id;
      END SetTarget;

    METHOD Target: DuraIdent;
      BEGIN
      RESULT ← TheTarget;
      END Target;

    METHOD GenericRecursionTarget: BOOLEAN;
      BEGIN
      RESULT ← TheGenericRecursionTarget;
      END GenericRecursionTarget;

    ⟨Redefining methods inherited from the YAFL compiler 448a⟩

  END GrammarExpression;
```

YAFL

Semantic analysis as redefined by `GrammarExpression` uses semantic analysis for identifiers as provided by the standard YAFL compiler, and ensures that the target for the assignment is a non-once attribute.

448a ⟨*Redefining methods inherited from the YAFL compiler* 448a⟩≡

```
REDEFINE METHOD Tag;
  BEGIN
  IF TheTarget ≠ Λ THEN
    TheTarget.UniqueTag;
    IF TheTarget.IsThis THEN
      TheGenericRecursionTarget ← TRUE;
      TheTarget ← Λ;
     ELSIF TheTarget.IsAttribute = Λ THEN
      Error (“Target must be an attribute”);
     ELSIF TheTarget.IsAttribute.Once THEN
      Error (“Target must be a non-ONCE attribute”);
      END;
    END;
  END Tag;
```

`SubTree` must be redefined by all non-terminals in a YAFL compiler to allow generic walkthroughs. The version displayed here only includes the part which is common to all `GrammarExpression` redefinitions of `SubTree`.

448b ⟨*Redefining methods inherited from the YAFL compiler* 448a⟩+≡

```
REDEFINE METHOD SubTree: ARRAY OF NonTerminal;
  BEGIN
  IF TheTarget ≠ Λ THEN
    RESULT.CREATE (1);
    RESULT_0 ← TheTarget;
    END;
  END SubTree;
```

Type checking for a grammatical expression translates into type checking for all its grammatical sub-expressions.

448c ⟨*Redefining methods inherited from the YAFL compiler* 448a⟩+≡

```
REDEFINE METHOD CheckType;
  BEGIN
  FOR p ∈ SubTree | p ≠ Λ DO
    p.UniqueCheckType;
    END;
  END CheckType;
```

`AggregateExpression` denotes expressions made of a number of sub-expressions connected by concatenation or disjunction.

449 ⟨`AggregateExpression` *class definition* 449⟩≡

```
CLASS AggregateExpression;
  INHERITS GrammarExpression(AggregateCodeGenerator);

  REDEFINE METHOD CREATE (LineNr, ColNr, Operator: INTEGER);
    PRE
      (Operator = LexicalAnalyzer.Dot) ∨
      (Operator = LexicalAnalyzer.Or);

  METHOD Append (Expr: GrammarExpression);
  METHOD Args: NTList(GrammarExpression);
  METHOD Operator: INTEGER;
    POST
      NO_CHANGE,
      (RESULT = LexicalAnalyzer.Dot) ∨
      (RESULT = LexicalAnalyzer.Or);
END AggregateExpression;
```

450 ⟨AggregateExpression *class implementation* 450⟩≡

```
CLASS AggregateExpression;
  INHERITS GrammarExpression(AggregateCodeGenerator);

  VAR
    TheList: NTList(GrammarExpression);
    TheOp: INTEGER;

  REDEFINE METHOD CREATE (LineNr, ColNr, Operator: INTEGER);
    BEGIN
    BASE(LineNr, ColNr);
    TheList.CREATE;
    TheOp ← Operator;
    END CREATE;

  ⟨AggregateExpression redefinitions from the YAFL compiler 451a⟩

  METHOD Append (Expr: GrammarExpression);
    BEGIN
    TheList.Append (Expr);
    Expr.SetFather (THIS);
    END Append;

  METHOD Args: NTList(GrammarExpression);
    BEGIN
    RESULT ← TheList;
    END Args;

  METHOD Operator: INTEGER;
    BEGIN
    RESULT ← TheOp;
    END Operator;

END AggregateExpression;
```

451a ⟨AggregateExpression *redefinitions from the YAFL compiler* 451a⟩≡

```
REDEFINE METHOD WhatAmI: ARRAY OF CHAR;
  BEGIN
  RESULT ← "AggregateExpression";
  END WhatAmI;

REDEFINE METHOD SubTree: ARRAY OF NonTerminal;
  BEGIN
  RESULT ← TheList.SubTree + BASE;
  END SubTree;

REDEFINE METHOD Tag;
  BEGIN
  BASE;
  TheList.UniqueTag;
  END Tag;
```

OptionalExpression is an expression made of an operand which is optional, as in:

```
GRAMMAR
  If Condition Then? Statement
```

where the keyword Then is optional.

451b ⟨OptionalExpression *class definition* 451b⟩≡

```
CLASS OptionalExpression;
  INHERITS GrammarExpression(OptionalCodeGenerator);

  REDEFINE METHOD CREATE (LineNr, ColNr: INTEGER;
                          Operand: GrammarExpression);

  METHOD Operand: GrammarExpression;
END OptionalExpression;
```

452a ⟨OptionalExpression *class implementation* 452a⟩≡

```
CLASS OptionalExpression;
  INHERITS GrammarExpression(OptionalCodeGenerator);

  VAR
    TheOperand: GrammarExpression;

  REDEFINE METHOD CREATE (LineNr,
                          ColNr: INTEGER;
                          Operand: GrammarExpression);
    BEGIN
    BASE(LineNr, ColNr);
    TheOperand ← Operand;
    Operand.SetFather (THIS);
    END CREATE;

  ⟨OptionalExpression redefinitions from the YAFL compiler 452b⟩

  METHOD Operand: GrammarExpression;
    BEGIN
    RESULT ← TheOperand;
    END Operand;

END OptionalExpression;
```

YAFL

452b ⟨OptionalExpression *redefinitions from the* Y^AF_L *compiler* 452b⟩≡

```
REDEFINE METHOD WhatAmI: ARRAY OF CHAR;
  BEGIN
  RESULT ← "OptionalExpression";
  END WhatAmI;

REDEFINE METHOD SubTree: ARRAY OF NonTerminal;
  BEGIN
  RESULT.CREATE (1);
  RESULT[0] ← TheOperand;
  RESULT ← RESULT + BASE;
  END SubTree;

REDEFINE METHOD Tag;
  BEGIN
  BASE;
  TheOperand.UniqueTag;
  END Tag;
```

`RepetitionExpression` denotes a grammatical expression which is made of repetition of an argument. `AtLeastOnce` indicates whether only non-empty repetitions are accepted.

`RepetitionExpression` also supports an optional *separator* that separates instances of the repeated argument, as in:

```
GRAMMAR
  LParent Expression + BY Comma RParent;
```

where the terminal `Comma` is used as separator between the `Expression` arguments.

453 ⟨`RepetitionExpression` *class definition* 453⟩≡

```
CLASS RepetitionExpression;
  INHERITS GrammarExpression(RepetitionCodeGenerator);

  REDEFINE METHOD CREATE (LineNr, ColNr: INTEGER;
                          Operand: GrammarExpression;
                          AtLeastOnce: BOOLEAN);

  METHOD AtLeastOnce: BOOLEAN;
  METHOD Operand: GrammarExpression;
  METHOD Separator: GrammarExpression;
  METHOD SetSeparator (Separator: GrammarExpression);

END RepetitionExpression;
```

454 ⟨RepetitionExpression *class implementation* 454⟩≡

```
CLASS RepetitionExpression;
  INHERITS GrammarExpression(RepetitionCodeGenerator);

  VAR
    TheAtLeastOnce: BOOLEAN;
    TheSeparator, TheOperand: GrammarExpression;

  REDEFINE METHOD CREATE (LineNr, ColNr: INTEGER;
                          Operand: GrammarExpression;
                          AtLeastOnce: BOOLEAN);
    BEGIN
    BASE(LineNr, ColNr);
    TheAtLeastOnce ← AtLeastOnce;
    TheOperand ← Operand;
    Operand.SetFather (THIS);
    END CREATE;

  ⟨RepetitionExpression redefinitions from the YAFL compiler 455a⟩

  METHOD Operand: GrammarExpression;
    BEGIN
    RESULT ← TheOperand;
    END Operand;

  METHOD AtLeastOnce: BOOLEAN;
    BEGIN
    RESULT ← TheAtLeastOnce;
    END AtLeastOnce;

  ⟨Separator handling methods 455b⟩

END RepetitionExpression;
```

455a ⟨`RepetitionExpression` *redefinitions from the YAFL compiler* 455a⟩≡

```
REDEFINE METHOD WhatAmI: ARRAY OF CHAR;
  BEGIN
  RESULT ← “RepetitionExpression”;
  END WhatAmI;

REDEFINE METHOD SubTree: ARRAY OF NonTerminal;
  BEGIN
  RESULT.CREATE (2);
  RESULT_0 ← TheOperand;
  RESULT_1 ← TheSeparator;
  RESULT ← RESULT + BASE;
  END SubTree;

REDEFINE METHOD Tag;
  BEGIN
  BASE;
  TheOperand.UniqueTag;
  IF TheSeparator ≠ Λ THEN
    TheSeparator.UniqueTag;
    END;
  END Tag;
```

455b ⟨*Separator handling methods* 455b⟩≡

```
METHOD Separator: GrammarExpression;
  BEGIN
  RESULT ← TheSeparator;
  END Separator;

METHOD SetSeparator (Separator: GrammarExpression);
  BEGIN
  TheSeparator ← Separator;
  END SetSeparator;
```

`AtomExpression` denotes the simplest form of grammatical expression, namely an identifier that denotes a lexeme (in which case `IsLexeme` returns a non-zero value) or a non-terminal class (in which case `IsClass` returns a non-void value).

456 ⟨`AtomExpression` *class definition* 456⟩≡

```
CLASS AtomExpression;
  INHERITS GrammarExpression(AtomCodeGenerator);

  REDEFINE METHOD CREATE (LineNr, ColNr: INTEGER;
                          Id: DuraIdent);
  METHOD Id: DuraIdent;
  METHOD IsLexeme: INTEGER;
  METHOD IsClass: DuraClassDefinition;
END AtomExpression;
```

457 ⟨AtomExpression *class implementation* 457⟩≡

```
CLASS AtomExpression;
  INHERITS GrammarExpression(AtomCodeGenerator);

  VAR
    TheId: DuraIdent;
    TheIsLexeme: INTEGER;
    TheIsClass: DuraClassDefinition;

  REDEFINE METHOD CREATE (LineNr, ColNr: INTEGER;
                          Id: DuraIdent);
    BEGIN
    BASE(LineNr, ColNr);
    TheId ← Id;
    END CREATE;

  ⟨AtomExpression redefinitions from the YAFL compiler 458a⟩

  METHOD Id: DuraIdent;
    BEGIN
    RESULT ← TheId;
    END Id;

  METHOD IsLexeme: INTEGER;
    BEGIN
    RESULT ← TheIsLexeme;
    END IsLexeme;

  METHOD IsClass: DuraClassDefinition;
    BEGIN
    RESULT ← TheIsClass;
    END IsClass;

END AtomExpression;
```

458a ⟨AtomExpression *redefinitions from the YAFL compiler* 458a⟩≡
```
REDEFINE METHOD WhatAmI: ARRAY OF CHAR;
  BEGIN
  RESULT ← "AtomExpression";
  END WhatAmI;

REDEFINE METHOD SubTree: ARRAY OF NonTerminal;
  BEGIN
  RESULT.CREATE (1);
  RESULT[0] ← TheId;
  RESULT ← RESULT + BASE;
  END SubTree;
```

YAFL

Semantic analysis on an `AtomExpression` starts by calling `UniqueTag` on `TheId`, as defined in the YAFL compiler to force the identifier to be attached to its corresponding declaration.

458b ⟨AtomExpression *redefinitions from the YAFL compiler* 458a⟩+≡
```
REDEFINE METHOD Tag;
  VAR
    TheType: Type;
  BEGIN
  BASE;
  TheId.UniqueTag;
```

Then, depending on what `TheId` is now attached to, various checks can be performed.

458c ⟨AtomExpression *redefinitions from the YAFL compiler* 458a⟩+≡
```
IF TheId.IsLexeme > 0 THEN
  TheIsLexeme ← TheId.IsLexeme;
 ELSIF TheId.IsAttribute ≠ Λ THEN
  ⟨The atom refers to an attribute, consider its type 458d⟩
 ELSIF TheId.IsClass ≠ Λ THEN
  TheIsClass ← TheId.IsClass;
 ELSE
  TheId.Error("Unexpected token class");
  END;
END Tag;
```

If `TheId` refers to an attribute, then `THIS` cannot already have a target attribute (See method *DuraExpressions.GrammarExpression.Target* on page 447).

458d ⟨*The atom refers to an attribute, consider its type* 458d⟩≡
```
IF Target ≠ Λ THEN
  Error ("Conflictual attribute assignments");
 ELSE
```

Otherwise, `SetTarget` is used to set `TheId` as the target attribute for the grammar expression denoted by `THIS` (See page 447).

459a ⟨*The atom refers to an attribute, consider its type* 458d⟩+≡
```
SetTarget (TheId);
```

and various simple checks can be performed, to ensure that the attribute's type is a valid non-terminal.

459b ⟨*The atom refers to an attribute, consider its type* 458d⟩+≡
```
TheType ← TheId.IsAttribute.GetType;
IF TheType.ArrayLevel > 0 THEN
  Error(``Unexpected array attribute");
 ELSIF TheType.SimpleType ≠ Λ THEN
  WHAT TheType.SimpleType.ClosestCanonic OF
    IN DuraClassDefinition:
      TheIsClass ← TAG;
      END;
   ELSE
    Error (``Does not refer to a valid non-terminal class");
    END;
  END;
END;
```

`CaseExpression` denotes a grammatical CASE expression, as described in page 261. It is made of a number of alternatives, implemented by the `CaseExpressionAlt` class, that form a disjunction, and a discriminating attribute that will receive a code to indicate which of the alternatives of the disjunction has been recognized.

459c ⟨*Grammar CASE construct class definitions* 459c⟩≡
```
CLASS CaseExpression;
  INHERITS GrammarExpression(CaseCodeGenerator);

  METHOD DiscriminantAttribute: SingleDataItem;
  METHOD Alts: NTList(CaseExpressionAlt);

END CaseExpression;
```

459d ⟨*Grammar CASE construct class definitions* 459c⟩+≡
```
CLASS CaseExpressionAlt;
  INHERITS NonTerminal(CaseAltCodeGenerator);

  METHOD DiscriminantValue: INTEGER;
  METHOD Expression: GrammarExpression;

END CaseExpressionAlt;
```

460a ⟨*Grammar CASE construct class implementations* 460a⟩≡

```
CLASS CaseExpression;
  INHERITS GrammarExpression(CaseCodeGenerator);

  VAR
    TheAttr: SingleDataItem;
    TheDiscr: DuraIdent;
    TheAlts: NTList(CaseExpressionAlt);

  ⟨CaseExpression redefinitions from the YAFL compiler 461b⟩

  METHOD DiscriminantAttribute: SingleDataItem;
    BEGIN
    RESULT ← TheAttr;
    END DiscriminantAttribute;

  METHOD Alts: NTList(CaseExpressionAlt);
    BEGIN
    RESULT ← TheAlts;
    END Alts;

  ⟨Parsing a CaseExpression 460b⟩

END CaseExpression;
```

Parsing a `CaseExpression` is rather predictable: parse the header, parse the alternative, and close the construction by accepting an `END` keyword.

460b ⟨*Parsing a* `CaseExpression` 460b⟩≡

```
REDEFINE METHOD Parse (Lkh: LookAhead);
  VAR
    TheAlt: CaseExpressionAlt;
  BEGIN
  WHAT Lkh OF
    IN DuraStreamLookAhead:
      TAG.Accept (LexicalAnalyzer.Case);
      TheDiscr ← TAG.AcceptDuraIdent;
      SetSon (TheDiscr);
      TAG.Accept (LexicalAnalyzer.Of);
      ⟨Parse the alternatives, set them into TheAlts 461a⟩
      IF TheAlts ≠ Λ THEN
        TheAlts.SetFather (THIS);
        END;
      Lkh.Skip (LexicalAnalyzer.End);
      END;
    END;
  END Parse;
```

Whenever we are in a state where an alternative can be found, we `CREATE` a new instance of `CaseExpressionAlt`, and let it parse itself (See method *DuraExpressions.CaseExpressionAlt.Parse* on page 463).

461a ⟨*Parse the alternatives, set them into* `TheAlts` 461a⟩≡
```
WHILE TAG.Ok ∧
      (TAG.CurrentToken ≠ TAG.End) DO
  TheAlt.CREATE(TAG.LineNr, TAG.ColNr);
  IF TheAlts = Λ THEN
    TheAlts.CREATE;
    END;
  TheAlts.Append (TheAlt);
  TheAlt.Parse (Lkh);
  END;
```
YAFL

461b ⟨`CaseExpression` *redefinitions from the* YAF_L *compiler* 461b⟩≡
```
REDEFINE METHOD SubTree: ARRAY OF NonTerminal;
  BEGIN
  RESULT.CREATE (1);
  RESULT_0 ← TheDiscr;
  IF TheAlts ≠ Λ THEN
    RESULT ← RESULT + TheAlts.SubTree;
    END;
  RESULT ← RESULT + BASE;
  END SubTree;

REDEFINE METHOD WhatAmI: ARRAY OF CHAR;
  BEGIN
  RESULT ← "CaseExpression";
  END WhatAmI;
```
YAFL

Semantic analysis first uses `UniqueTag` as provided by the YAF_L compiler, to use the visibility rules of the language to attach `TheDiscr` to the corresponding declaration.

461c ⟨`CaseExpression` *redefinitions from the* YAF_L *compiler* 461b⟩+≡
```
REDEFINE METHOD Tag;
  BEGIN
  BASE;
  TheDiscr.UniqueTag;
  ⟨Make sure that TheDiscr refers to non-once an integer attribute 462a⟩
  IF TheAlts ≠ Λ THEN
    TheAlts.UniqueTag;
    END;
  END Tag;
```

TheDiscr is not just a plain YAFL identifier, it is an instance of a DURA-specialized identifier (See page 673), that provides a convenient access to the attribute it refers to *if it indeed refers to an attribute !* (See method *DuraIdentifiers.DuraIdent.IsAttribute* on page 678). If TheDiscr does not refer to an attribute, TheDiscr.IsAttribute yields VOID.

462a ⟨*Make sure that* TheDiscr *refers to non-once an integer attribute* 462a⟩≡

```
TheAttr ← TheDiscr.IsAttribute;
IF (TheAttr = Λ) ∨ (TheAttr.GetType.ArrayLevel ≠ 0) ∨
     ¬ TheAttr.GetType.
       SimpleType.Match (PredefItems.Integer) THEN
  TheDiscr.Error (''A CASE expression discriminant must be" +
                  '' an integer attribute");
  END;
```

A CaseExpressionAlt is made of a discriminant expression TheDiscrExpr *that must evaluate to an integer*, that will be stored in TheDiscrValue; and a grammatical expression argument in TheExpr. CaseExpressionAlts are gathered by disjunction in alternatives, and the different TheDiscrValues of these alternatives will allow us to know which of the alternatives has been recognized.

462b ⟨*Grammar CASE construct class implementations* 460a⟩+≡

```
CLASS CaseExpressionAlt;
  INHERITS NonTerminal(CaseAltCodeGenerator);

  VAR
    TheDiscrExpr: TypedNonTerminal;
    TheDiscrValue: INTEGER;
    TheExpr: GrammarExpression;

  ⟨CaseExpressionAlt redefinitions from the YAFL compiler 463b⟩

  METHOD DiscriminantValue: INTEGER;
    BEGIN
    RESULT ← TheDiscrValue;
    END DiscriminantValue;

  METHOD Expression: GrammarExpression;
    BEGIN
    RESULT ← TheExpr;
    END Expression;

  ⟨Parsing a CaseExpressionAlt 463a⟩

  ⟨CheckType as redefined by CaseExpressionAlt 464a⟩

END CaseExpressionAlt;
```

When parsing a `CaseExpressionAlt`, we make use of specific features that have been added to `LookAhead` when inheriting in `DuraStreamLookAhead` (See page 809). By means of creators (See class *DuraCreators.DuraStreamCreator* on page 405) we know that all `LookAhead` instances that will be created while running `DURA` will be of the `DuraStreamLookAhead` class. However, when redefining a method, the types of its arguments must remain exactly the same. Therefore, we must cast the argument `Lkh` into a `DuraStreamLookAhead` using an explicit `WHAT` statement.

463a ⟨*Parsing a* `CaseExpressionAlt` 463a⟩≡
```
REDEFINE METHOD Parse (Lkh: LookAhead);
  BEGIN
  WHAT Lkh OF
    IN DuraStreamLookAhead:
      TheDiscrExpr ← TAG.AcceptPlainExpr;
      SetSon (TheDiscrExpr);
      Lkh.Accept (LexicalAnalyzer.Colon);
      TheExpr ← TAG.AcceptGrammarExpression;
      SetSon (TheExpr);
      Lkh.Skip (LexicalAnalyzer.SemiColon);
      Lkh.Skip (LexicalAnalyzer.End);
      Lkh.Skip (LexicalAnalyzer.SemiColon);
      END;
    END;
  END Parse;
```

463b ⟨`CaseExpressionAlt` *redefinitions from the* $\mathcal{Y}\!A\!F_L$ *compiler* 463b⟩≡
```
REDEFINE METHOD WhatAmI: ARRAY OF CHAR;
  BEGIN
  RESULT ← "CaseExpressionAlt";
  END WhatAmI;

REDEFINE METHOD SubTree: ARRAY OF NonTerminal;
  BEGIN
  RESULT.CREATE (2);
  RESULT_0 ← TheDiscrExpr;
  RESULT_1 ← TheExpr;
  END SubTree;

REDEFINE METHOD Tag;
  BEGIN
  TheDiscrExpr.UniqueTag;
  TheExpr.UniqueTag;
  END Tag;
```

The discriminant expression must be a compile-time evaluable constant. However, there is no structural difference in YAFL between a plain and a constant expression. It is constant folding, by means of the `GetFolded` method, that indicates whether an expression can be evaluated statically. YAFL

464a ⟨`CheckType` *as redefined by* `CaseExpressionAlt` 464a⟩≡

```
REDEFINE METHOD CheckType;
   BEGIN
   BASE;
   IF TheDiscrExpr.GetFolded = Λ THEN
      TheDiscrExpr.Error (‘‘Constant expression expected”);
    ELSE
      WHAT TheDiscrExpr.GetFolded OF
         IN IntegerLiteral:
            TheDiscrValue ← TAG.Value;
            END;
       ELSE
         TheDiscrExpr.Error (‘‘Integer expression expected”);
         END;
      END;
   END CheckType;
```

464b ⟨`TerminalSetExpression` *class definition* 464b⟩≡

```
CLASS TerminalSetExpression;
   INHERITS GrammarExpression(TerminalSetCodeGenerator);

   METHOD GetTerminalSet: TerminalSet;

END TerminalSetExpression;
```

`TerminalSetExpression` denotes a grammar expression that matches a set of tokens, as in:

```
GRAMMAR
  Expression [Plus,Minus,Star,Slash] Expression;
```

which is of course equivalent to:

```
GRAMMAR
  Expression (Plus OR Minus OR Star OR Slash) Expression;
```

but which can produce a far more efficient parser. A versatile syntax can be used to specify the set of tokens `TheTermExpr` (See page 467) within a `TerminalSetExpression`.

465a ⟨`TerminalSetExpression` *class implementation* 465a⟩≡

```
CLASS TerminalSetExpression;
  INHERITS GrammarExpression(TerminalSetCodeGenerator);

  VAR
    TheTermExpr: TermExpr;

  ⟨TerminalSetExpression redefinitions from the YAFL compiler 465b⟩

  METHOD GetTerminalSet: TerminalSet;
    BEGIN
    RESULT ← TheTermExpr.GetTerminalSet;
    END GetTerminalSet;

  ⟨Parsing a TerminalSetExpression 466⟩

END TerminalSetExpression;
```

YAFL

465b ⟨`TerminalSetExpression` *redefinitions from the YAFL compiler* 465b⟩≡

```
REDEFINE METHOD WhatAmI: ARRAY OF CHAR;
  BEGIN
  RESULT ← ‘‘TerminalSetExpression”;
  END WhatAmI;

REDEFINE METHOD Tag;
  BEGIN
  BASE;
  TheTermExpr.UniqueTag;
  END Tag;

REDEFINE METHOD SubTree: ARRAY OF NonTerminal;
  BEGIN
  RESULT.CREATE(1);
  RESULT0 ← TheTermExpr;
  END SubTree;
```

The grammar for token sets is:

$$
\begin{array}{lcl}
\mathit{TermExpr} & ::= & [\ T_1\ ] \\
T_1 & ::= & T_2\ | \\
 & & T_1\ ,\ T_2\ | \\
 & & T_1\ -\ T_2\ | \\
 & & T_1\ T_2 \\
T_2 & ::= & (\ T_1\ )\ | \\
 & & *\ | \\
 & & \mathit{token} \\
 & & \mathit{token}\ \mathit{TO}\ \mathit{token}
\end{array}
$$

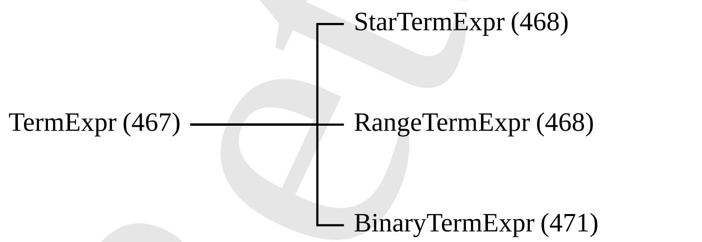
The following lengthy method implements the recursive descent to recognize this grammar. It builds (yet another !) expression tree, in the form of `TermExpr`-derived classes (See the next page).

466 ⟨*Parsing a* `TerminalSetExpression` 466⟩≡

```
REDEFINE METHOD Parse (Lkh: LookAhead);

  METHOD AcceptTerminalTerm: TermExpr;
    VAR
      TheStarExpr: StarTermExpr;
      TheRangeExpr: RangeTermExpr;
    BEGIN
    IF Lkh.CurrentToken = Lkh.LeftParen THEN
      Lkh.GetToken;
      RESULT ← AcceptTermExpr;
      Lkh.Accept (Lkh.RightParen);
     ELSIF Lkh.CurrentToken = Lkh.Star THEN
      TheStarExpr.CREATE (Lkh.LineNr, Lkh.ColNr);
      RESULT ← TheStarExpr;
      Lkh.GetToken;
     ELSE
      TheRangeExpr.CREATE(Lkh.LineNr, Lkh.ColNr);
      TheRangeExpr.Parse(Lkh);
      RESULT ← TheRangeExpr;
      END;
    END AcceptTerminalTerm;

  METHOD AcceptTermExpr: TermExpr;
    VAR
      GoOn: BOOLEAN;
      BinExpr : BinaryTermExpr;
    BEGIN
    RESULT ← AcceptTerminalTerm;
    GoOn ← TRUE;
    WHILE GoOn DO
```

```
      CASE Lkh.CurrentToken OF
        Lkh.Comma:
          Lkh.GetToken;
          BinExpr.CREATE(Lkh.LineNr, Lkh.ColNr);
         BinExpr.Set (RESULT, Addition ← TRUE, AcceptTerminalTerm);
          RESULT ← BinExpr;
          END;
        Lkh.Ident:
          BinExpr.CREATE(Lkh.LineNr, Lkh.ColNr);
         BinExpr.Set (RESULT, Addition ← TRUE, AcceptTerminalTerm);
          RESULT ← BinExpr;
          END;
        Lkh.Minus:
          Lkh.GetToken;
          BinExpr.CREATE(Lkh.LineNr, Lkh.ColNr);
         BinExpr.Set (RESULT, Addition ← FALSE, AcceptTerminalTerm);
          RESULT ← BinExpr;
          END;
       ELSE
        GoOn ← FALSE;
        END;
      END;
    END AcceptTermExpr;

  BEGIN
  Lkh.Accept (LexicalAnalyzer.LeftBracket);
  TheTermExpr ← AcceptTermExpr;
  Lkh.Accept (LexicalAnalyzer.RightBracket);
  END Parse;
```

Figure 7.6 Inheritance tree for terminal set expressions



`TermExpr` is the base class for all terminal set expressions. The only significant service it provides is the `GetTerminalSet` method that returns a `TerminalSet` (See page 558) and that must be redefined by derived class.

468a ⟨*Terminal expression intermediate classes* 468a⟩≡

```
CLASS TermExpr;
  INHERITS NonTerminal(GC);

  DEFERRED METHOD GetTerminalSet: TerminalSet;

  REDEFINE METHOD CheckType;
    BEGIN
    END CheckType;

END TermExpr;
```

`StarTermExpr` denotes the set of all terminals, usually used in cases such as `[* - SemiColon]` to indicate any token except the `SemiColon` are accepted.

To build the corresponding `TerminalSet`, `GetTerminalSet` creates an empty one, and negates it using the method `Negate` as provided for the bitmaps as defined in the YAFL standard library, producing a full one, and that's it !

468b ⟨*Terminal expression intermediate classes* 468a⟩+≡

```
CLASS StarTermExpr;
  INHERITS TermExpr;

  REDEFINE METHOD GetTerminalSet: TerminalSet;
    BEGIN
    RESULT.CREATE;
    RESULT.Negate;
    END GetTerminalSet;

  REDEFINE METHOD SubTree: ARRAY OF NonTerminal;
    BEGIN
    END SubTree;

  REDEFINE METHOD WhatAmI: ARRAY OF CHAR;
    BEGIN
    RESULT ← "StarTermExpr";
    END WhatAmI;

END StarTermExpr;
```

`RangeTermExpr` denotes an interval of terminals, as in `[Array TO While]`. In a degenerate form, it is also used to denote a single token.

It contains two couples of attributes: `FirstId` and `LastId` are the identifiers that denote the begin and the end of the range, while `FirstCode` and `LastCode` denote the corresponding token codes.

469 ⟨*Terminal expression intermediate classes* 468a⟩+≡

```
CLASS RangeTermExpr;
  INHERITS TermExpr;

  VAR
    FirstId, LastId: DuraIdent;
    FirstCode, LastCode: INTEGER;

  ⟨Parsing a RangeTermExpr 470a⟩
  ⟨Tagging a RangeTermExpr 470b⟩

  REDEFINE METHOD GetTerminalSet: TerminalSet;
    BEGIN
    RESULT.CREATE;
    FOR i ∈ FirstCode .. LastCode DO
      RESULT.Set(i);
      END;
    END GetTerminalSet;

  REDEFINE METHOD SubTree: ARRAY OF NonTerminal;
    BEGIN
    RESULT.CREATE (2);
    RESULT_0 ← FirstId;
    RESULT_1 ← LastId;
    END SubTree;

  REDEFINE METHOD WhatAmI: ARRAY OF CHAR;
    BEGIN
    RESULT ← "RangeTermExpr";
    END WhatAmI;

END RangeTermExpr;
```

When parsing a range, one first recognizes the first identifier. If the current token is `TO`, one skips it and recognizes the second identifier of the range.

470a ⟨*Parsing a* `RangeTermExpr` 470a⟩≡

```
REDEFINE METHOD Parse(Lkh: LookAhead);

  METHOD AcceptDuraIdent(Lkh: LookAhead): DuraIdent;
    BEGIN
    WHAT Lkh OF
      IN DuraStreamLookAhead:
        RESULT ← TAG.AcceptDuraIdent;
        END;
      END;
    END AcceptDuraIdent;

  BEGIN
  FirstId ← AcceptDuraIdent (Lkh);
  IF Lkh.CurrentToken = Lkh.To THEN
    Lkh.GetToken;
    LastId ← AcceptDuraIdent (Lkh);
    END;
  END Parse;
```

YAFL

470b ⟨*Tagging a* `RangeTermExpr` 470b⟩≡

```
REDEFINE METHOD Tag;

  ⟨Method CheckId to make sure that an ident is a lexeme 471a⟩

  BEGIN
  FirstCode ← CheckId (FirstId);
  LastCode ← CheckId (LastId);
  IF LastCode ≤ 0 THEN
    LastCode ← FirstCode;
    END;
  END Tag;
```

`CheckId` uses `UniqueTag` from the standard $\text{Y}^{\text{A}}\text{F}_{\text{L}}$ compiler to attach the identifier to its corresponding declaration according to the visibility rules of the language, then uses `IsLexeme` (See page 678) to check whether `Id` refers to a lexeme.

471a ⟨*Method* `CheckId` *to make sure that an ident is a lexeme* 471a⟩≡

```
METHOD CheckId (Id: DuraIdent): INTEGER;
  BEGIN
  IF Id ≠ Λ THEN
    Id.UniqueTag;
    RESULT ← Id.IsLexeme;
    IF RESULT ≤ 0 THEN
      Id.Error(“Lexeme identifier expected”);
      END;
    END;
  END CheckId;
```

`BinaryTerm` implements a binary expression, made of two sub-expressions in `TheLeft` and `TheRight`, and a flag `TheAddition` that indicates whether the resulting `TerminalSet` should be the union or the set difference of the terminal sets built out of its arguments.

471b ⟨*Terminal expression intermediate classes* 468a⟩+≡

```
CLASS BinaryTermExpr;
  INHERITS TermExpr;

  VAR
    TheLeft, TheRight: TermExpr;
    TheAddition: BOOLEAN;

  METHOD Set (Left: TermExpr;
              Addition: BOOLEAN;
              Right: TermExpr);
    BEGIN
    TheLeft ← Left;
    TheAddition ← Addition;
    TheRight ← Right;
    END Set;

  ⟨BinaryTermExpr redefinitions from the YAFL compiler 472c⟩

  ⟨Computing GetTerminalSet for BinaryTermExpr 472a⟩


END BinaryTermExpr;
```

472a ⟨*Computing* `GetTerminalSet` *for* `BinaryTermExpr` 472a⟩≡

```
REDEFINE METHOD GetTerminalSet: TerminalSet;
  VAR
    RightSet: TerminalSet;
  BEGIN
  RESULT ← TheLeft.GetTerminalSet;
  RightSet ← TheRight.GetTerminalSet;
  IF TheAddition THEN
    RESULT.Or(RightSet);
   ELSE
    RightSet.Negate;
    RESULT.And (RightSet);
    ⟨Ensure the set difference worked 472b⟩
    END;
  END GetTerminalSet;
```

472b ⟨*Ensure the set difference worked* 472b⟩≡

```
DEBUG
  RightSet ← TheRight.GetTerminalSet;
  ASSERT ∀ i ∈ 0 .. RightSet.Size - 1 •
    RightSet_i ⇒ ¬ RESULT_i;
  END;
```

YAFL

472c ⟨`BinaryTermExpr` *redefinitions from the YAFL compiler* 472c⟩≡

```
REDEFINE METHOD SubTree: ARRAY OF NonTerminal;
  BEGIN
  RESULT.CREATE (2);
  RESULT_0 ← TheLeft;
  RESULT_1 ← TheRight;
  END SubTree;

REDEFINE METHOD Tag;
  BEGIN
  TheLeft.UniqueTag;
  TheRight.UniqueTag;
  END Tag;

REDEFINE METHOD WhatAmI: ARRAY OF CHAR;
  BEGIN
  RESULT ← "BinaryTermExpr";
  END WhatAmI;
```

473a ⟨*Definition import list* 473a⟩≡
```
FROM DuraIdentifiers IMPORT DuraIdent;
FROM YaflParser IMPORT NonTerminal;
FROM DuraClDefinition IMPORT DuraClassDefinition;
FROM DuraCG IMPORT AggregateCodeGenerator,
                   RepetitionCodeGenerator,
                   OptionalCodeGenerator,
                   IdentCodeGenerator, AtomCodeGenerator,
                   CaseCodeGenerator,
                   CaseAltCodeGenerator,
                   TerminalSetCodeGenerator,
                   GrammarExprCodeGenerator;
FROM DuraFlat IMPORT TerminalSet;
FROM YaflClasses IMPORT ClassDeclaration;
FROM YaflDeclarations IMPORT SingleDataItem;
FROM YaflGCode IMPORT GC, DummyNTCodeGenerator;
FROM YaflLex IMPORT LexicalAnalyzer;
FROM YaflNTList IMPORT NTList;
```

473b ⟨*Implementation import list* 473b⟩≡
```
FROM List IMPORT List;
FROM Streams IMPORT StdErr;

IMPORT LookAhead;
FROM YaflClasses IMPORT ClassDeclaration;
FROM YaflGCode IMPORT GC;
FROM YaflLiteral IMPORT IntegerLiteral;
FROM YaflNTList IMPORT NTList;
FROM YaflPredefined IMPORT PredefItems;
FROM YaflType IMPORT TypedNonTerminal, Type;

IMPORT DuraCfg;
FROM DuraClImplementation IMPORT DuraClassImplementation;
FROM DuraLookAhead IMPORT DuraStreamLookAhead;
```

7.13 DuraFlat

The `DuraFlat` module implements the flat representation for grammar rules, as well as a number of operations on these flattened rules. Even if there seems to be little reason for doing so, there is nothing essential that prevents from using this module independently from `DURA` , for instance using `yacc` or a `yacc`-like tool.

A flat representation is made of simple grammar rules $N ::= \mu$ where $\mu \in (N \cup T)*$, but for implementation reasons, they are structured differently. Instead of having:

$$\begin{array}{lcl} N & ::= & \mu_1 \\ N & ::= & \mu_2 \\ & & \vdots \\ N & ::= & \mu_n \end{array}$$

we have:

$$\begin{array}{lcl} N & ::= & \mu_1 \mid \\ & & \mu_2 \mid \\ & & \vdots \\ & & \mu_n \end{array}$$

Such a compound construction is said to be a *rule*, implemented by the `FlatRule` class (See page 480). $\mu_1 \ldots \mu_n$ are said to be *right hand sides* or more shortly RHS's, implemented by the `FlatRHS` (See page 513) class.

Rather than concentrating on the sole `DURA` issue, I tried to make these primitive as relevant as possible for other uses or even for experimenting alternate designs for `DURA`. Hence, rather than providing a single, optimized primitive that would be strictly `DURA`-specific, this module provides several more general services which combination might as well not be optimal in terms of performance or resource usage.

```
⟨Definition ⟩≡
  DEFINITION MODULE DuraFlat;

  ⟨Definition import list 560c⟩

     ⟨FlatRule class definition 480⟩

     ⟨FlatRHS class definition 513⟩
        ⟨RHSList class definition 539b⟩

     ⟨FlatRuleComponent class definition 540⟩
        ⟨FlatTerminal class definition 544c⟩
        ⟨FlatTerminalSet class definition 548⟩
        ⟨FlatNonTerminal class definition 554a⟩

     ⟨TerminalSet class definition 558b⟩
     ⟨NonTerminalSet class definition 560a⟩

  END DuraFlat;

⟨Implementation ⟩≡
  IMPLEMENTATION MODULE DuraFlat;

  ⟨Implementation import list 561⟩

     ⟨FlatRule class implementation 481⟩

     ⟨FlatRHS class implementation 514⟩
        ⟨RHSList class implementation 539c⟩

     ⟨FlatRuleComponent class implementation 541b⟩
        ⟨FlatTerminal class implementation 545a⟩
        ⟨FlatTerminalSet class implementation 549a⟩
        ⟨FlatNonTerminal class implementation 554b⟩

     ⟨TerminalSet class implementation 559a⟩
     ⟨NonTerminalSet class implementation 560b⟩

  END DuraFlat;
```

7.13.1 Flattened grammar rules

A `FlatRule` is a flattened grammar rule, in the sense defined in the introduction to this chapter. It is implemented as inheriting from the list of its attached right hand sides.

Please note that some of these `FlatRules` will refer to non-terminal classes as defined in enriched YAFL source code, while some other will be *synthetic* grammar rules, generated automatically in the flattening process.

For instance, consider the following code fragment:

```
NONTERMINAL CLASS OrderClause;
  INHERITS BaseNonTerminal;

  GRAMMAR
    Order By? Expression;
```

assuming that $Order$ and By are terminals, while $Expression$ is a visible non-terminal class (possibly using an import clause). This grammar rule basically expresses the fact that the By keyword is optional.

As such, this representation does not conform to the flattened representation. Hence, it is transformed into the following:

$$\begin{array}{lcl} OrderClause & ::= & Order\ Opt_{By}\ Expression \\ Opt_{By} & ::= & By\ | \\ & & \epsilon \end{array}$$

[2] $OrderClause$ is attached to a non-terminal, while Syn_{By} is a computational artefact.

Before we go even further, we define $Symbol$, which is $N \cup T$, the union of terminal ($FlatRule$) and non-terminal ($TokenCode$, (See page 14)) symbols. Non-terminal symbols are represented here as references to $FlatRule$s:

$NullableLevelType$ is an enumerated type that will be used to qualify the nullability of various elements ($FlatRule$, $FlatRHS$, etc.).

$$NullableLevelType == Error \mid AlwaysNull \mid Nullable \mid NotNullable$$

$Symbol$ is the direct transcription of $N \cup T$ in the common language theory terminology, that is, terminals and non-terminals, that can occur within a grammar rule.

$$\begin{array}{|l} Symbol = TokenCode \cup \rho FlatRule \\ SymbolString = \mathrm{seq}\ Symbol \end{array}$$

[2] The names given to rules are phony, of course. They are added to help understanding, while in practice, the flattened representation mainly deals with numbers.

FlatRule

In the actual implementation listed hereunder, the RHS attribute is implemented using an inheritance relationship. This is one of the very few cases within this project where implementation inheritance is used. Implementation inheritance qualifies a situation where inheritance is used in a way that does not reflect a true polymorphic nature, but which implements a convenient data structure concatenation and more convenient notations. In most cases, implementation inheritance is used as a convenient substitute for aggregation.

A flat rule contains a number of right hand sides, implemented by the $FlatRHS$ class (See page 511). Based on these right hand sides, most of the rule's properties can be deduced easily: nullability level, various attached sets, etc.

$RHS : \mathbb{P}\ FlatRHS$

$RHS \neq \phi$
$\forall\, r \in RHS \bullet \delta r.Enclosing = self$

$Strings$ is the set of strings that can be produced by a partial – understand: where some non-terminal references remain – or complete – with terminals only – derivation:
$s \in f.Strings \Leftrightarrow f \overset{*}{\Rightarrow} s$.
$TerminalStrings$ is $Strings$ restricted to complete derivations only.

$Strings, TerminalStrings : \mathbb{P}\ SymbolString$ (See the preceding page)

$Strings = \bigcup_{r \in RHS} r.Strings$
$TerminalStrings = \{x \in Strings \mid \operatorname{ran} x \subset TokenCode\}$

The production set attached to a $FlatRule$ f is a set of *references* to flat rules (hence the ρ). See page 979 for more information about ObjectZ references as used in this document.

$ProductionSet : \mathbb{P}\,\rho FlatRule$
$Cycle, Reducible : \mathbb{B}$

$ProductionSet = \bigcup_{r \in RHS} r.ProductionSet$

The constraint given above explains how $ProductionSet$ is computed. We can also express $ProductionSet$ as the set of non-terminals that can be produced by that the $FlatRule$ at hand. Formally, this can be expressed as:

$p \in ProductionSet \Leftrightarrow \langle p \rangle \in Strings$

p is not a valid $SymbolString$, but the sequence $\langle p \rangle$ which contains a single element p is.

$Cycle$ indicates the presence of a grammatical cycle, that is, a nonterminal N such that $N \stackrel{+}{\Rightarrow} N$.

$Cycle = \text{true} \Leftrightarrow \exists\, r \in ProductionSet \bullet \delta r = self$

A $FlatRule$ is reducible if it can derive to a finite sequence of terminals.

$Reducible = \text{true} \Leftrightarrow TerminalStrings \neq \phi$

$NullableLevel : NullableLevelType$ (See page 476)

$NullableLevel = AlwaysNull \Leftrightarrow \forall\, r \in RHS \bullet$
$\quad r.NullableLevel = AlwaysNull$
$NullableLevel = Nullable \Leftrightarrow \exists\, r \in RHS \bullet$
$\quad r.NullableLevel = NullableNullableLevel = NotNullable \Leftrightarrow \forall\, r \in RHS \bullet$
$\quad r.NullableLevel = NotNullable$

The following attributes yield sets of $TokenCode$ (See page 14), that is, sets of terminal symbols.

$MiddleSet$ yields the set of $TokenCode$s that can occur within the productions of a grammar rule, that is, not as first or last symbol, but somewhere between these extremities.

$MiddleSet : \mathbb{P}\ TokenCode$

$MiddleSet = \bigcup_{r \in RHS} r.MiddleSet$

Given a non-empty sequence s, $tail\ s$ returns the subsequence made of s without the leading element, while $front\ s$ returns the subsequence made of s without the trailing element. $front\ tail\ s$ yields the sequence made of s without the leading and trailing element. Z sequences are defined formally as partial functions. Hence, given a sequence l, $\mathrm{ran}\ l$ returns the set of the elements contained in l. It is a way of transforming a sequence where ordering is significant into its canonically defined set where order is unimportant.

$MiddleSet = \bigcup_{s \in TerminalStrings \mid \#s > 0} \mathrm{ran}\ front\ tail\ s$

$FollowSet$ is a function that yields FOLLOW set of a $FlatRule$, given a context – the grammar – in the form of a set of $FlatRule$s, in which the rule occurs. $PrecedeSet$ is the logical counterpart which returns the PRECEDE set.

$FollowSet,$
$PrecedeSet : \mathbb{P}\ FlatRule \mapsto \mathbb{P}\ TokenCode$ (See page 14)

$$
\begin{aligned}
&\forall\, Grammar \in \mathbb{P}\ FlatRule,\ t \in TokenCode \bullet\\
&\quad t \in FollowSet(Grammar) \Leftrightarrow\\
&\qquad \exists\, r \in Grammar \bullet\\
&\qquad\quad \exists\, s \in r.Strings \bullet\\
&\qquad\qquad \#s \geq 2 \land \exists\, i \in 1..\#s - 1 \bullet\\
&\qquad\qquad\quad s(i) \in \rho FlatRule \land \delta s(i) = self \land s(i+1) = t
\end{aligned}
$$

$$
\begin{aligned}
&\forall\, Grammar \in \mathbb{P}\ FlatRule,\ t \in TokenCode \bullet\\
&\quad t \in PrecedeSet(Grammar) \Leftrightarrow\\
&\qquad \exists\, r \in Grammar \bullet\\
&\qquad\quad \exists\, s \in r.Strings \bullet\\
&\qquad\qquad \exists\, i \in 1..\#s - 1 \bullet\\
&\qquad\qquad\quad s(i+1) \in \rho FlatRule \land \delta s(i+1) = self \land s(i) = t
\end{aligned}
$$

Unlike $FollowSet$ and $PrecedeSet$ and just like $MiddleSet$ above, $FirstSet$ only depends on the rule's own derivations, and can be more defined as a plain attribute rather than as a function.

$$FirstSet : \mathbb{P}\ TokenCode$$

$$FirstSet = \bigcup_{s \in TerminalStrings} \{head\ s\}$$

and finally, $LastSet$:

$$LastSet : \mathbb{P}\ TokenCode$$

$$LastSet = \bigcup_{s \in TerminalStrings} \{last\ s\}$$

480 ⟨`FlatRule` *class definition* 480⟩≡

```
CLASS FlatRule;
  INHERITS ArrayList(FlatRHS);

  ⟨Attaching a nonterminal class to a rule, definition 483a⟩
  ⟨FlatReference method definition 483c⟩
  ⟨Dump method definition 485b⟩
  ⟨Closure management for FlatRule, definition 491c⟩
  ⟨Checking a FlatRule's nullability, definition 488⟩
  ⟨Checking a FlatRule's reduceability, definition 491a⟩
  ⟨Can THIS be inlined ?, definition 484⟩

  ⟨Handle associativity and precedence, definition 492b⟩
  ⟨Numbering methods, definition 486c⟩
  ⟨Goto code generation, definition 500a⟩
  ⟨Goto array computation, definition 503b⟩
  ⟨Usage counting methods, definition 487a⟩
  ⟨Handling transitions on me, definition 498b⟩

  ⟨Attached sets (FIRST, FOLLOW etc.), definition 504a⟩

  ⟨Comparing with another FlatRule, definition 509a⟩

  ⟨Generating equivalent yacc source, definition 509c⟩
  ⟨Computing the rule's production set, definition 507b⟩

END FlatRule;
```

481 ⟨`FlatRule` *class implementation* 481⟩≡

```
CLASS FlatRule;
  INHERITS ArrayList(FlatRHS);

  ⟨Attaching a nonterminal class to a rule, implementation 483b⟩

  ⟨FlatReference method implementation 483d⟩

  ⟨Dump method implementation 485c⟩
  ⟨Checking a FlatRule's reduceability, implementation 491b⟩
  ⟨Checking a FlatRule's nullability, implementation 489a⟩

  ⟨Numbering methods, implementation 486d⟩
  ⟨Usage counting methods, implementation 487b⟩
  ⟨Closure management for FlatRule 491d⟩

  ⟨Making sure that no VOID right hand side is appended 482⟩
  ⟨Can THIS be inlined ?, implementation 485a⟩
  ⟨Handle associativity and precedence, implementation 492c⟩

  ⟨Goto code generation, implementation 500b⟩
  ⟨Goto array computation, implementation 503c⟩
  ⟨Handling transitions on me, implementation 499a⟩
  ⟨Attached sets (FIRST, FOLLOW etc.), implementation 505⟩
  ⟨Comparing with another FlatRule, implementation 509b⟩
  ⟨Generating equivalent yacc source, implementation 510a⟩
  ⟨Computing the rule's production set, implementation 508⟩

END FlatRule;
```

The following chunk has been added in order to find the origin of a bug, which symptom was the apperance of `VOID` elements in the `FlatRule`.

It is very typical of the YAFL way of bug tracking: instead of spending hours in a runtime debugger to inspect why and where such `VOID` elements may have been appended, alter the source code through redefinition with assertions in such a way that a reasonably comprehensive error message will be issued if a `VOID` element is appended.

One of the nicest property of this method lies in the fact that if ever an entirely different error occurs months later, and has the same effect, namely, to append a `VOID` right hand side to a rule, it will be catched flawlessly, without having to go through yet another phase of lengthy and tedious runtime debugging. Since the assertions are removed from the production system, and induce no performance penalty at all, they are left in the source code. It serves a documentation purpose, as well as improving quality in the longer term. Amazingly enough, in [14], Hanson actually *removed* all assertions from the source code of lcc before publishing it ! This is obviously a major difference in attitude: Hanson considers assertions as a mere debugging aid, while I consider them as a way of describing a system's property at any given time with absolute accuracy.

These redefinitions also serve another purpose, namely, attaching the `FlatRHS` to the `FlatRule` denoted by `THIS`.

482 ⟨*Making sure that no* `VOID` *right hand side is appended* 482⟩≡

```
REDEFINE METHOD Append (El: FlatRHS);
  BEGIN
  ASSERT El ≠ Λ;
  BASE (El);
  El.AttachToFlatRule (THIS);
  END Append;

REDEFINE METHOD Insert (Pos: INTEGER;
                        El: FlatRHS);
  BEGIN
  ASSERT El ≠ Λ;
  BASE (Pos, El);
  El.AttachToFlatRule (THIS);
  END Insert;
```

The `SetClass` method can be used to indicate that a flattened grammar rule refers to a user-defined non-terminal represented by a class. If this attached class is not set, the rule will be known to be *synthetic*, that is, generated automatically as intermediate grammar rule within another rule.

Some `FlatRules` won't have an attached class. The grammar rule which is defined within a class can turn into more than a single `FlatRule`, and only the outermost `FlatRule` will be attached to its enclosing class, thereby indicating that whenever one of the rule's RHS is reduced, a new instance of its attached class must be created.

For instance, in the example:

```
NONTERMINAL CLASS OrderClause;
  INHERITS BaseNonTerminal;

  GRAMMAR
    Order By? Expression;
```

which somehow translates in a flat representation of the form:

$$
\begin{array}{lcl}
\mathit{OrderClause} & ::= & \mathit{Order}\ \mathit{Opt}_{By}\ \mathit{Expression} \\
\mathit{Opt}_{By} & ::= & \mathit{By}\ | \\
 & & \epsilon
\end{array}
$$

$\mathit{OrderClause}$ will have an attached class, while the internal, synthetic rule Opt_{By} won't.

483a ⟨*Attaching a nonterminal class to a rule, definition* 483a⟩≡

```
METHOD SetClass (Class: DuraClassDefinition);
METHOD Class: DuraClassDefinition;
```

483b ⟨*Attaching a nonterminal class to a rule, implementation* 483b⟩≡

```
VAR
  TheClass: DuraClassDefinition;

METHOD SetClass (Class: DuraClassDefinition);
  BEGIN
  TheClass ← Class;
  END SetClass;

METHOD Class: DuraClassDefinition;
  BEGIN
  RESULT ← TheClass;
  END Class;
```

The `FlatReference` returns a reference to the rule it is being applied to (See page 554). In other words, this reference, when used within a grammar rule, will refer to the rule referred to by THIS.

483c ⟨`FlatReference` *method definition* 483c⟩≡

```
METHOD FlatReference: FlatNonTerminal;
```

483d ⟨`FlatReference` *method implementation* 483d⟩≡

```
METHOD FlatReference: FlatNonTerminal;
  BEGIN
  RESULT.CREATE;
  RESULT.SetRule (THIS);
  END FlatReference;
```

7.13.1.1 Inlining grammar rules

We'll say that the rule can be inlined if three conditions are fulfilled:

- It is used in only one place. This condition must be checked *after that unreachable rules have been removed*. Hence, the rule cannot be directly recursive. A rule that is reachable from the root grammar rule is at least referenced by another rule than itself. If the rule is recursive, it includes a reference to itself, it is then used in at least two places, and consequently, it cannot be inlined.
- There must be only one right hand side defined for the rule.
- The reduction, if any, that is attached to this right hand side must be reducible, that is, removable without altering the system's semantics.

This last condition can be best shown in the following example. Consider the classes:

```
NONTERMINAL CLASS A;
  VAR
    b: B;
  GRAMMAR
    b;  -- implicitely referring to B, without inheritance
END A;

NONTERMINAL CLASS B;
  VAR
    c: C;
  GRAMMAR
    c;  -- implicitely referring to C, without inheritance
END B;

NONTERMINAL CLASS C;
  GRAMMAR
    Ident;
END C;
```

the corresponding grammar rules are:

$$\begin{aligned} A &::= B \\ B &::= C \\ C &::= \mathit{Ident} \end{aligned}$$

While grammatically correct, reducing this to $A ::= \mathit{Ident}$ or even $A ::= C$ would be semantically and typewise wrong. Such automatic reductions are correct in case of grammar rules that are generated from inheritance rules, but are definitely wrong when they are stated explicitely in the grammar, as in the example above. $A ::= \mathit{Ident}$ cannot be fetched properly into a A instance.

484 ⟨*Can* THIS *be inlined ?, definition* 484⟩≡
  METHOD Inlinable: FlatRHS;

485a ⟨*Can* THIS *be inlined ?, implementation* 485a⟩≡

```
METHOD Inlinable: FlatRHS;
  VAR
    RHS: FlatRHS;
  BEGIN
  IF Size = 1 THEN
    RHS ← First;
    IF (UsageCount = 1) ∨ (RHS.Size = 1) THEN
      IF (RHS.Reduction = Λ) THEN
        RESULT ← RHS;
        END;
      END;
    END;
  END Inlinable;
```

485b ⟨Dump *method definition* 485b⟩≡

```
METHOD Dump (Output: OutputStream);
```

485c ⟨Dump *method implementation* 485c⟩≡

```
METHOD Dump (Output: OutputStream);
  BEGIN
  DumpHeader (Output);
  FOR RHS ∈ THIS DO
    Output.WriteInt (RHS.Size, 0);
    Output.WriteString ("> ");
    RHS.Dump (Output);
    END;
  IF (FirstSet ≠ Λ) ∧ (FirstSet.UsedBits > 0) THEN
    Output.WriteLine ("FIRST:");
    Output.WriteString ("  ");
    FirstSet.Dump(Output);
    END;
  IF (LastSet ≠ Λ) ∧ (LastSet.UsedBits > 0) THEN
    Output.WriteLine ("LAST:");
    Output.WriteString ("  ");
    LastSet.Dump(Output);
    END;
  IF (FollowSet ≠ Λ) ∧ (FollowSet.UsedBits > 0) THEN
    Output.WriteLine ("FOLLOW:");
    Output.WriteString ("  ");
    FollowSet.Dump(Output);
    END;
  ⟨Dump the production set 486a⟩
  Output.WriteLn;
  END Dump;
```

Some day, perhaps.

486a ⟨*Dump the production set* 486a⟩≡

```
IF (ProductionSet ≠ Λ) ∧ (ProductionSet.UsedBits > 0) THEN
  -- To do...
  END;
```

486b ⟨Dump *method implementation* 485c⟩+≡

```
METHOD DumpHeader (Output: OutputStream);
  BEGIN
  IF Nr > 0 THEN
    Output.WriteInt (Nr, 0);
    END;
  IF TheClass ≠ Λ THEN
    Output.WriteString (" (" + TheClass.DotImage + ")");
    END;
  IF AlwaysNull THEN
    Output.WriteString (" <ALWAYS NULL>");
   ELSIF NullableLevel = CodeUnknownLevel THEN
    Output.WriteString (" <UNKNOWN>");
   ELSIF Nullable THEN
    Output.WriteString (" <NULLABLE>");
   ELSE
    Output.WriteString (" <NOT NULLABLE>");
    END;
  Output.WriteLine (" :-");
  END DumpHeader;
```

Since all this will turn into an automaton, we need numbers, hence the possibility of numbering virtually everything.

486c ⟨*Numbering methods, definition* 486c⟩≡

```
METHOD Nr: INTEGER;
METHOD SetNr (Nr: INTEGER);
```

486d ⟨*Numbering methods, implementation* 486d⟩≡

```
VAR
  TheNr: INTEGER;

METHOD SetNr (Nr: INTEGER);
  BEGIN
  TheNr ← Nr;
  END SetNr;

METHOD Nr: INTEGER;
  BEGIN
  RESULT ← TheNr;
  END Nr;
```

In some cases, it can be useful to know how many times a given rule is used in a specification. The following chunk implements a rather naive scheme, where one can reset the counter, increment it, and consult it afterwards.

The usage counters are computed when removing unused grammar rules. Since this is a rather efficient operation, or, more exactly, an operation that would be quite easy to optimize, it is perfectly reasonable to call the `RemoveUnreachableRules` method (See page 565) before using these counters just to be sure they are up to date.

487a ⟨*Usage counting methods, definition* 487a⟩≡

```
METHOD ClearUsageCount;
METHOD IncUsageCount;
METHOD UsageCount: INTEGER;
```

487b ⟨*Usage counting methods, implementation* 487b⟩≡

```
VAR
  TheUsageCount: INTEGER;

METHOD ClearUsageCount;
  BEGIN
  TheUsageCount ← 0;
  END ClearUsageCount;

METHOD IncUsageCount;
  BEGIN
  TheUsageCount ← TheUsageCount + 1;
  END IncUsageCount;

METHOD UsageCount: INTEGER;
  BEGIN
  RESULT ← TheUsageCount;
  END UsageCount;
```

`Nullable` indicates whether a given flat rule can produce an empty string, while `AlwaysNull` indicates that the rule cannot produce anything else than the empty string.

This behaviour is implemented by an integer attribute that can take five values, each representing a state:

CodeErrorLevel Nullability has still not been investigated for this rule

CodeUnknownLevel The rule's nullability is currently being computed

CodeAlwaysNullLevel The rule is always null

CodeNullableLevel The rule can be null, but can also produce non-empty strings

CodeNotNullableLevel The rule cannot produce an empty string

Testing whether a given grammar rule is nullable cannot happen locally. A global algorithm takes an entire syntactical specification into account to decide whether a grammar rule is nullable or not.

488 ⟨*Checking a* `FlatRule`*'s nullability, definition* 488⟩≡

```
METHOD Nullable: BOOLEAN;
METHOD AlwaysNull: BOOLEAN;

CONST
  CodeErrorLevel = 0;
  CodeUnknownLevel = 1;
  CodeAlwaysNullLevel = 2;
  CodeNullableLevel = 3;
  CodeNotNullableLevel = 4;

METHOD SetNullableLevel(Level: INTEGER);
METHOD NullableLevel: INTEGER;
METHOD ComputeNullableLevel;
```

489a ⟨*Checking a* `FlatRule`*'s nullability, implementation* 489a⟩≡

```
VAR
  TheNullableLevel: INTEGER;

METHOD Nullable: BOOLEAN;
  BEGIN
  RESULT ← NullableLevel ≠ CodeNotNullableLevel;
  END Nullable;

METHOD AlwaysNull: BOOLEAN;
  BEGIN
  RESULT ← NullableLevel = CodeAlwaysNullLevel;
  END AlwaysNull;

METHOD SetNullableLevel(Level: INTEGER);
  BEGIN
  TheNullableLevel ← Level;
  END SetNullableLevel;

METHOD NullableLevel: INTEGER;
  BEGIN
  RESULT ← TheNullableLevel;
  END NullableLevel;
```

489b ⟨*Checking a* `FlatRule`*'s nullability, implementation* 489a⟩+≡

```
METHOD ComputeNullableLevel;
  VAR
    NewLevel: INTEGER;
    IsNullable, IsAlwaysNull, IsUnknown: BOOLEAN;
  BEGIN
  IsAlwaysNull ← TRUE;
  ⟨First, go through the RHS's 490a⟩
```

The loop in this chunk computes the following formulas:

$$\begin{aligned}
&IsNullable = \exists\, Rhs \in THIS \bullet Rhs.NullableLevel = CodeAlwaysNullLevel \lor \\
&\qquad\qquad Rhs.NullableLevel = CodeNullableLevel \\
&IsAlwaysNull = \forall\, Rhs \in THIS \bullet Rhs.NullableLevel = CodeAlwaysNullLevel \\
&IsUnknown = \exists\, Rhs \in THIS \bullet Rhs.NullableLevel = CodeUnknownLevel
\end{aligned}$$

but is slightly more efficient by performing all computations within a single loop.

490a ⟨*First, go through the RHS's* 490a⟩≡

```
FOR RHS ∈ THIS WHILE ¬ IsUnknown DO
  CASE RHS.NullableLevel OF
    CodeUnknownLevel:
      IsUnknown ← TRUE;
      END;
    CodeAlwaysNullLevel:
      IsNullable ← TRUE;
      END;
    CodeNullableLevel:
      IsNullable ← TRUE;
      IsAlwaysNull ← FALSE;
      END;
    CodeNotNullableLevel:
      IsAlwaysNull ← FALSE;
      END;
   ELSE
    END; -- Case
  END;   -- For
```

Then, depending on what was found, we set `THIS`'s nullability.

490b ⟨*Checking a* `FlatRule`*'s nullability, implementation* 489a⟩+≡

```
NewLevel ← CodeUnknownLevel;
IF ¬ IsUnknown THEN
  IF IsNullable THEN
    IF IsAlwaysNull THEN
      NewLevel ← CodeAlwaysNullLevel;
     ELSE
      NewLevel ← CodeNullableLevel;
      END;
   ELSE
    NewLevel ← FlatRule.CodeNotNullableLevel;
    END;
  END;
SetNullableLevel (NewLevel);
END ComputeNullableLevel;
```

Reducability is quite similar to nullability, for that matter. The same global algorithm includes the logic...

491a ⟨*Checking a* `FlatRule`*'s reduceability, definition* 491a⟩≡

```
METHOD Reducible: BOOLEAN;
METHOD SetReducible (Flag: BOOLEAN);
```

491b ⟨*Checking a* `FlatRule`*'s reduceability, implementation* 491b⟩≡

```
VAR
  TheReducible: BOOLEAN;

METHOD Reducible: BOOLEAN;
  BEGIN
  RESULT ← TheReducible;
  END Reducible;

METHOD SetReducible (Flag: BOOLEAN);
  BEGIN
  TheReducible ← Flag;
  END SetReducible;
```

491c ⟨*Closure management for* `FlatRule`*, definition* 491c⟩≡

```
METHOD ComputeClosure: INTEGER;
METHOD Closure: FlatItemSet;
```

The closure of a `FlatRule` is made of all the items that can start the rule. It is computed rather simply, by computing the union of the closures of each starting item of each RHS within the rule.

Closure management is a global issue, that is, it can only be computed globally on an entire grammar. `ComputeClosure` is called repeatedly by `ComputeClosures` (See page 567) until all closures of all known grammar rules are stable.

491d ⟨*Closure management for* `FlatRule` 491d⟩≡

```
VAR
  TheClosure: FlatItemSet;

METHOD Closure: FlatItemSet;
  BEGIN
  RESULT ← TheClosure;
  END Closure;
```

`ComputeClosure` compute `THIS`'s closure according to the current state, that is, the currenly computed closures within the system.

It returns an integer that indicate a total number of bits, so that the routine that computes the closures loops, and recomputes all closures until the system is stable, that is, until recomputing the closures does not add a new entry to any of these closure sets.

The closure of a `FlatRule` is the union of the closures of the first items (starting with index 0, as YAFL arrays and collection classes do) of all its RHS's.

492a ⟨*Closure management for* `FlatRule` 491d⟩+≡

```
METHOD ComputeClosure: INTEGER;
  BEGIN
  IF TheClosure = Λ THEN
    TheClosure.CREATE;
    END;
  FOR Rhs ∈ THIS DO
    RESULT ← RESULT + Rhs.ComputeClosure;
    TheClosure.Or(Rhs.GetItem(0).Closure);
    END;
  RESULT ← RESULT + TheClosure.UsedBits;
  END ComputeClosure;
```

7.13.1.2 Associativity and precedence

492b ⟨*Handle associativity and precedence, definition* 492b⟩≡

```
METHOD HandleAssociativity;
```

492c ⟨*Handle associativity and precedence, implementation* 492c⟩≡

```
METHOD HandleAssociativity;
  VAR
    MaxPrecedence,
    MinPrecedence: INTEGER;
    RHSArray: ARRAY OF FlatRHS;

    ⟨Recursive method Dispatch 494⟩

  BEGIN
```

If the grammar rule is made of a single RHS, there is no possible associativity and precedence handling. For precedence to make sense, one must deal with recursive grammar rules. If we have a recursive grammar rule of the form $N ::= \alpha N \beta$ with $\alpha, \beta \in (N \cup T)*$, N can never be reduced, in absence of an alternative rule to exit from recursion.

493a ⟨*Handle associativity and precedence, implementation* 492c⟩+≡
```
IF Size > 1 THEN
   ⟨First, sort the right hand sides according to their precedence 493b⟩
   MaxPrecedence ← First.CanonicPrecedence;
   MinPrecedence ← Last.CanonicPrecedence;
   IF MinPrecedence ≠ MaxPrecedence THEN
      ⟨There are at least two different precedences defined 493e⟩
      END;
   END;
END HandleAssociativity;

⟨Comparison method for sorting 493d⟩
```

By the virtues of code reusability in an object-oriented world, sorting a `List` can be performed by merely calling the `Sort` method. The only prerequisite lies in the fact that the `ShouldSwap` method is redefined (see below).

493b ⟨*First, sort the right hand sides according to their precedence* 493b⟩≡
```
Sort;
```

The following chunk ensures that the sorting method worked properly,

493c ⟨*First, sort the right hand sides according to their precedence* 493b⟩+≡
```
ASSERT ∀i ∈ 0 .. Size - 2 •
   THIS_i.CanonicPrecedence ≥ THIS_{i+1}.CanonicPrecedence;
```

493d ⟨*Comparison method for sorting* 493d⟩≡
```
REDEFINE METHOD ShouldSwap (One, Two: FlatRHS): BOOLEAN;
   BEGIN
   RESULT ← One.CanonicPrecedence < Two.CanonicPrecedence;
   END ShouldSwap;
```

Well, this is where it all happens... First, `RHSArray` receives an array made of all the elements (RHS) in `THIS` (remember that `FlatRule` inherits from `AbstractList`). `RHSArray` will be used as a buffer while the new grammar rules are built using the precedence and associativity information attached to the grammar.

493e ⟨*There are at least two different precedences defined* 493e⟩≡
```
RHSArray ← Row;
```

We then `Purge` the grammar rule, that is, remove all the elements from the underlying `AbstractList`. The `Dispatch` method that actually performs the true work will use `RHSArray`.

493f ⟨*There are at least two different precedences defined* 493e⟩+≡
```
Purge;
Dispatch (RHSArray, THIS);
```

The `Dispatch` method takes an array of right hand sides, and a rule. It will fetch the right hand sides on the rules as long as they all have the same precedence. If some of the right hand sides have a smaller precedence[3], a new rule is created, and the right hand sides with the smaller precedence are fetched onto this new rule using a recursive call to `Dispatch`.

Let's consider the very typical example, sorted by decreasing precedence:

$$
\begin{array}{rcll}
E & ::= & E + E \mid & \text{Left associative, priority 5} \\
 & & E - E \mid & \text{Left associative, priority 5} \\
 & & E * E \mid & \text{Left associative, priority 3} \\
 & & E / E \mid & \text{Left associative, priority 3} \\
 & & E^E \mid & \text{Right associative, priority 1} \\
 & & (E) \mid & \\
 & & digit &
\end{array}
$$

494 ⟨*Recursive method* `Dispatch` 494⟩≡

```
METHOD Dispatch (RHSArray: ARRAY OF FlatRHS;
                 Rule: FlatRule);
  VAR
    Pos,
    MaxPrecedence: INTEGER;
    NewRule: FlatRule;
    RHS: FlatRHS;

    ⟨PatchRHS utility method 498a⟩

  BEGIN
  MaxPrecedence ← RHSArray_0.CanonicPrecedence;
  Pos ← 1;
  ⟨Compute Pos as the number of FlatRHSs with the maximum precedence 495b⟩
  FOR i ∈ 0 .. Pos - 1 DO
    Rule.Append (RHSArray_i);
    END;
```

[3] We assume that they cannot have a higher precedence since the array must is sorted by construction.

`Rule` then gets the rules extracted from the original grammar rule that have the highest precedence, and adds a new synthetic rule E'

$$\begin{aligned} E \ ::= \ & E + E \mid \\ & E - E \mid \\ & E' \end{aligned}$$

495a ⟨*Recursive method* `Dispatch` 494⟩+≡

```
IF Pos < RHSArray.SIZE THEN
  NewRule.CREATE;
  ⟨Patch some FlatRHS from Rule according to their associativity 497⟩
  RHS.CREATE;
  Rule.Append (RHS);
  RHS.Append (NewRule.FlatReference);
  Dispatch (RHSArray.SLICE (Pos, RHSArray.SIZE - Pos),
            NewRule);
  END;
END Dispatch;
```

495b ⟨*Compute* `Pos` *as the number of* `FlatRHS`*s with the maximum precedence* 495b⟩≡

```
WHILE (Pos < RHSArray.SIZE) ∧
      (RHSArray_Pos.CanonicPrecedence = MaxPrecedence) DO
  Pos ← Pos + 1;
  END;
```

The following chunk takes a subset of the right hand sides of a rule which have all the same precedence, and patches the rules according to their associativity to remove ambiguities. It also replaces references to the main grammar rule (`THIS`) by intermediate grammar rules.

For instance, considering the original grammar:

$$\begin{aligned} E \ ::= \ & E + E \mid && \text{– Left associative, priority 5} \\ & E - E \mid && \text{– Left associative, priority 5} \\ & E * E \mid && \text{– Left associative, priority 3} \\ & E / E \mid && \text{– Left associative, priority 3} \\ & E ** E \mid && \text{– Right associative, priority 1} \\ & (E) \mid && \\ & digit && \end{aligned}$$

Splitting it into chunks with the same precedence, and taking associativity into account yields successively:

$$\begin{aligned} E \ ::= \ & E + E \mid \\ & E - E \mid \\ & E_1 \end{aligned}$$

Then, after that $E+E$ and $E-E$ have been patched according to their associativity:

$$
\begin{aligned}
E \ ::= \ & E + E_1 \mid \\
& E - E_1 \mid \\
& E_1
\end{aligned}
$$

Then, similarly:

$$
\begin{aligned}
E \ ::= \ & E + E_1 \mid \\
& E - E_1 \mid \\
& E_1 \\
E_1 \ ::= \ & E * E \mid \\
& E / E \mid \\
& E_2
\end{aligned}
$$

and:

$$
\begin{aligned}
E \ ::= \ & E + E_1 \mid \\
& E - E_1 \mid \\
& E_1 \\
E_1 \ ::= \ & E_1 * E_2 \mid \\
& E_1 / E_2 \mid \\
& E_2
\end{aligned}
$$

Well, let's continue:

$$
\begin{aligned}
E \ ::= \ & E + E_1 \mid \\
& E - E_1 \mid \\
& E_1 \\
E_1 \ ::= \ & E_1 * E_2 \mid \\
& E_1 / E_2 \mid \\
& E_2 \\
E_2 \ ::= \ & E * * E \mid \\
& E_3
\end{aligned}
$$

and, considering that "$**$" is right-associative:

$$
\begin{aligned}
E \ ::= \ & E + E_1 \mid \\
& E - E_1 \mid \\
& E_1 \\
E_1 \ ::= \ & E_1 * E_2 \mid \\
& E_1 / E_2 \mid \\
& E_2 \\
E_2 \ ::= \ & E_3 * * E2 \mid \\
& E_3
\end{aligned}
$$

and finally:

$$
\begin{array}{lcl}
E & ::= & E + E_1 \mid \\
  &   & E - E_1 \mid \\
  &   & E_1 \\
E_1 & ::= & E_1 * E_2 \mid \\
  &   & E_1 / E_2 \mid \\
  &   & E_2 \\
E_2 & ::= & E_3 ** E2 \mid \\
  &   & E_3 \\
E_3 & ::= & (E) \mid \\
  &   & digit
\end{array}
$$

which is not ambiguous, and implements associativity and precedence according to the original specification.

This approach differs from `yacc`'s, which uses precedence in the LR automaton to choose between a shift and a reduce depending on the lookahead token. Due to the support for backtracking, our automaton is already very complex, and we preferred dealing with this issue at an earlier stage by performing changes on the grammar. This approach suffers from one drawback, namely the fact that the number of reductions for a rule in a priority grammar can go as high as the number of different priorities, while `yacc`, in similar cicumstances, requires a single reduction.

Table 8.1 on page 944 shows the performance of the `DURA`-generated parser for languages with an abnormally high number of different precedences (namely, ObjectZ).

497 ⟨*Patch some* `FlatRHS` *from* `Rule` *according to their associativity* 497⟩≡

```
FOR r ∈ Rule DO
  RHS ← r.Canonical;
  CASE RHS.Associativity OF
    InterRule.LeftAssoc:
      PatchRHS (RHS, 0, Rule);
      PatchRHS (RHS, RHS.Size-1, NewRule);
      END;
    InterRule.RightAssoc:
      PatchRHS (RHS, 0, NewRule);
      PatchRHS (RHS, RHS.Size-1, Rule);
      END;
   ELSE
    PatchRHS (RHS, 0, NewRule);
    PatchRHS (RHS, RHS.Size-1, NewRule);
    END;
  END;
```

PatchRHS takes a right hand side, a position and a flattened grammar rule Rule. If the component at the given position is a non-terminal component that refers to THIS (which qualifies recursion), it is replaced by a non-terminal component that refers to Rule (See method *DuraFlat.FlatRule.FlatReference* on page 483).

498a ⟨PatchRHS *utility method* 498a⟩≡

```
METHOD PatchRHS (RHS: FlatRHS;
                 Pos: INTEGER;
                 Rule: FlatRule);
  BEGIN
  WHAT RHS_Pos OF
    IN FlatNonTerminal:
      ASSERT (TAG.Rule ≠ THIS) ⇒ (TAG.Rule.Nr ≠ Nr);
      IF (TAG.Rule = THIS) THEN
        RHS.Delete (Pos);
        RHS.Insert(Pos, Rule.FlatReference);
        RHS_Pos;
        END;
      END;
   ELSE
    END;
  END PatchRHS;
```

7.13.2 Transitions on a non-terminal

What we mean here by "*transitions on me*" are transitions that consume a non-terminal reduced by the rule denoted by THIS. For instance, when a transitions goes from state 1 to state 2 by consuming an Expression, this transition must be included in the TransitionsOnMe attached to the flat rule attached to Expression.

498b ⟨*Handling transitions on me, definition* 498b⟩≡

```
METHOD SetTransitionsOnMe (TransitionsOnMe:
                                NonTerminalTransitionList);
METHOD TransitionsOnMe: NonTerminalTransitionList;
METHOD UniqueTargetOnMe: PDAState;
```

499a ⟨*Handling transitions on me, implementation* 499a⟩≡

```
VAR
  TheTransitionsOnMe : NonTerminalTransitionList;

METHOD SetTransitionsOnMe (TransitionsOnMe:
                              NonTerminalTransitionList);
  BEGIN
  TheTransitionsOnMe ← TransitionsOnMe;
  END SetTransitionsOnMe;

METHOD TransitionsOnMe: NonTerminalTransitionList;
  BEGIN
  RESULT ← TheTransitionsOnMe;
  END TransitionsOnMe;
```

The purpose of the following chunk is to determine whether when performing a reduce on the non-terminal denoted by `THIS`, the push-down automaton can only end up in a single state, in which case `UniqueTargetOnMe` returns that state. Otherwise, it returns `VOID`.

499b ⟨*Handling transitions on me, implementation* 499a⟩+≡

```
VAR
  TheUniqueTargetOnMe: PDAState;
  TheUniqueTargetAsked: BOOLEAN;

METHOD UniqueTargetOnMe: PDAState;
  VAR
    State: PDAState;
  BEGIN
  IF ¬ TheUniqueTargetAsked THEN
    TheUniqueTargetAsked ← TRUE;
    IF TransitionsOnMe ≠ Λ THEN
      ⟨Determining whether a single target can be reached 499c⟩
      END;
    END;
  RESULT ← TheUniqueTargetOnMe;
  END UniqueTargetOnMe;
```

We use a quantifier to check whether all transitions in `TransitionsOnMe` actually go to the same target state.

499c ⟨*Determining whether a single target can be reached* 499c⟩≡

```
State ← TransitionsOnMe.First.Two;
IF ∀t ∈ TransitionsOnMe •
    t.Two = State  THEN
  TheUniqueTargetOnMe ← State;
  END;
```

7.13.3 Code generation

500a ⟨*Goto code generation, definition* 500a⟩≡
```
METHOD GotoMethodName: ARRAY OF CHAR;
METHOD GenerateGotoMethodCall (Output: DuraOutputStream;
                               Depth: INTEGER);
METHOD GenerateGotoMethodBody (Output: DuraOutputStream);
```

Every non-terminal has a GOTO method which, given the state the parser was in *before* starting to analyze the non-terminal, returns the state the parser must move to after reduction. Such a GOTO method is only generated for compiled parsers, interpreted parsers manage this state transition issue based on tables.

The GOTO method name is derived from the numbering scheme for `FlatRule` (See method *DuraFlat.FlatRule.Nr* on page 486).

500b ⟨*Goto code generation, implementation* 500b⟩≡
```
METHOD GotoMethodName: ARRAY OF CHAR;
  BEGIN
  RESULT ← ‘‘Goto_” + IntConversions.IntToString (Nr, 0);
  END GotoMethodName;
```

`GenerateGotoMethodCall` must generate a YAFL expression that computes the result of a GOTO on the non-terminal denoted by `THIS`. Three cases are considered:

- If reducing the non-terminal denoted by `THIS` always moves the parser to a single target state, this state number is pasted. The detection of this case is based on `UniqueTargetOnMe` (See page 499).
- If we are dealing with an interpreted parser, we generate a call to `GenericGoto` (See page 874) that will compute the target state based on tables
- If we are dealing with a compiled parser, we generate a call to the method named after `GotoMethodName` (see above).

501 ⟨*Goto code generation, implementation* 500b⟩+≡

```
METHOD GenerateGotoMethodCall (Output: DuraOutputStream;
                               Depth: INTEGER);
  VAR
    State: PDAState;
  BEGIN
  State ← UniqueTargetOnMe;
  IF State ≠ Λ THEN
    Output.WriteInt (State.Nr, 0);
   ELSIF DuraCfg.GenerateInterpretedParsers THEN
    Output.WriteString (‘‘GenericGoto (”);
    Output.WriteInt (Nr, 0);
    Output.WriteString (‘‘,PreviousState(”);
    Output.WriteInt (Depth, 0);
    Output.WriteString (‘‘))”);
   ELSE
    Output.WriteString (GotoMethodName + ‘‘(PreviousState(”);
    Output.WriteInt (Depth, 0);
    Output.WriteString (‘‘))”);
    END;
  END GenerateGotoMethodCall;
```

GenerateGotoMethodBody is only called for compiled parsers. It takes a DuraOutputStream (See page 859) as parameter, which provides several YAFL-specific formatting facilities, such as indentation.

502a ⟨*Goto code generation, implementation* 500b⟩+≡
```
METHOD GenerateGotoMethodBody (Output: DuraOutputStream);
  VAR
    Name: ARRAY OF CHAR;
    First, Last, TargetNr: INTEGER;
    TheList: NonTerminalTransitionList;
  BEGIN
  Name ← GotoMethodName;
  IF UniqueTargetOnMe ≠ Λ THEN
    Output.WriteLine (``-- No method " + Name + ``: single target");
   ELSE
    Output.WriteLine (``METHOD " + Name +
                          ``(PreviousState: INTEGER): INTEGER;");
    Output.IncIndent (2);
    Output.WriteLine (``BEGIN");
    ⟨Gather all relevant transitions 502b⟩
    IF TheList ≠ Λ THEN
      Output.WriteLine (``CASE PreviousState OF");
      Output.IncIndent (2);
      First ← 0;
      WHILE First < TheList.Size DO
       ⟨Consider transitions on me starting at First 503a⟩
        First ← Last;
        END;
      Output.End;
      END;
    Output.WriteLine (``END " + Name + ``;");
    Output.DecIndent (2);
    END;
  Output.WriteLn;
  END GenerateGotoMethodBody;
```

The relevant transition are all the transitions on which an instance of the non-terminal denoted by THIS is accepted. This set of transitions if computed by TransitionsOnMe (See page 498).

502b ⟨*Gather all relevant transitions* 502b⟩≡
```
TheList ← TransitionsOnMe;
```

503a ⟨*Consider transitions on me starting at* `First` 503a⟩≡

```
Last ← First + 1;
TargetNr ← TheList_First.Two.Nr;
WHILE (Last < TheList.Size) ∧
      (TheList_Last.Two.Nr = TargetNr) DO
  Last ← Last + 1;
  END;
FOR i ∈ First .. Last−1 DO
  IF i ≠ First THEN
    Output.WriteChar (“,”);
    END;
  Output.WriteInt (TransitionsOnMe_i.One.Nr, 0);
  END;
Output.WriteLine (“:”);
Output.IncIndent (2);
Output.WriteString (“RESULT := ”);
Output.WriteInt (TargetNr, 0);
Output.WriteLine (“;”);
Output.End;
```

`GotoArray` generates an array of integers for the $Goto$ function attached to `THIS`. We have: $RESULT_i = j, j \neq 0$ if there is a transition from state i to state j by accepting an instance of the non-terminal denoted by `THIS`. If $RESULT_i$ = 0, there is no transition starting at state i that consumes an instance of the non-terminal denoted by `THIS`.

As described here, the GOTO table are not compressed. In order to reduce the memory requirement of the parsers, the GOTO tables are optimized globally (See method *DuraFlSpec.FlatSpecification.StoreGotoTable* on page 583).

503b ⟨*Goto array computation, definition* 503b⟩≡

```
METHOD GotoArray(AutomatonSize: INTEGER): ARRAY OF INTEGER;
```

503c ⟨*Goto array computation, implementation* 503c⟩≡

```
METHOD GotoArray(AutomatonSize: INTEGER): ARRAY OF INTEGER;
  BEGIN
  RESULT.CREATE (AutomatonSize);
  FOR t ∈ TransitionsOnMe DO
    RESULT_t.One.Nr ← t.Two.Nr;
    END;
  END GotoArray;
```

7.13.4 Attached sets

The following chunks manage several sets of terminals that can be attached to a non-terminal. $FIRST$, $FOLLOW$ are described extensively in the literature (). These sets, together with $MIDDLE$, $LAST$ and $PRECEDE$ are defined formally in the ObjectZ specification in page 477.

Except for *FOLLOW*, these sets are not used by DURA as it stands now. They have been included for the sake of completeness, and because I have plans to implement an alternate error recovery scheme better suited for some classes of language, which would be based on *LAST* and *MIDDLE* rather than on *FOLLOW*.

504a ⟨*Attached sets (*FIRST, FOLLOW *etc.), definition* 504a⟩≡

```
METHOD FirstSet: TerminalSet;
METHOD LastSet: TerminalSet;
METHOD FollowSet: TerminalSet;
METHOD PrecedeSet: TerminalSet;
METHOD MiddleSet: TerminalSet;

METHOD SetFirstSet (TheSet: TerminalSet);
METHOD SetLastSet (TheSet: TerminalSet);
METHOD SetFollowSet (TheSet: TerminalSet);
METHOD SetPrecedeSet (TheSet: TerminalSet);
METHOD SetMiddleSet (TheSet: TerminalSet);
```

504b ⟨*Attached sets (*FIRST, FOLLOW *etc.), definition* 504a⟩+≡

```
METHOD ComputeAttachedSets;
```

505 ⟨*Attached sets (*FIRST, FOLLOW *etc.), implementation* 505⟩≡

```
VAR
  TheFirstSet,
  TheLastSet,
  TheFollowSet,
  ThePrecedeSet,
  TheMiddleSet: TerminalSet;

METHOD FirstSet: TerminalSet;
  BEGIN
  RESULT ← TheFirstSet;
  END FirstSet;

METHOD LastSet: TerminalSet;
  BEGIN
  RESULT ← TheLastSet;
  END LastSet;

METHOD FollowSet: TerminalSet;
  BEGIN
  RESULT ← TheFollowSet;
  END FollowSet;

METHOD PrecedeSet: TerminalSet;
  BEGIN
  RESULT ← ThePrecedeSet;
  END PrecedeSet;

METHOD MiddleSet: TerminalSet;
  BEGIN
  RESULT ← TheMiddleSet;
  END MiddleSet;
```

506 ⟨*Attached sets (*FIRST, FOLLOW *etc.), implementation* 505⟩+≡

```
METHOD SetFirstSet (TheSet: TerminalSet);
  BEGIN
  TheFirstSet ← TheSet;
  END SetFirstSet;

METHOD SetLastSet (TheSet: TerminalSet);
  BEGIN
  TheLastSet ← TheSet;
  END SetLastSet;

METHOD SetFollowSet (TheSet: TerminalSet);
  BEGIN
  TheFollowSet ← TheSet;
  END SetFollowSet;

METHOD SetPrecedeSet (TheSet: TerminalSet);
  BEGIN
  ThePrecedeSet ← TheSet;
  END SetPrecedeSet;

METHOD SetMiddleSet (TheSet: TerminalSet);
  BEGIN
  TheMiddleSet ← TheSet;
  END SetMiddleSet;
```

The rule is not responsible for the complete computation of its attached set. As in many other occurrences in the grammar constructs managed by DURA, computing these sets in a reliable way requires a global approach, rather than a local one.

`ComputeAttachedSets`, as defined here, compute these sets using the current state of the system, that is, what is known at a given moment. It will be called repeatedly by `FlatSpec` (See page 302) until none of the sets attached to non-terminals of the grammar evolve.

507a ⟨*Attached sets (*FIRST, FOLLOW *etc.), implementation* 505⟩+≡

```
METHOD ComputeAttachedSets;
  BEGIN
  FOR RHS ∈ THIS | RHS.Size > 0 DO
    RHS.First.AddToFirstSet (THIS);
    RHS.Last.AddToLastSet (THIS);
    RHS.ComputeMiddleSet;
    RHS.ComputeAttachedSets;
    END;
  IF Nullable THEN
    FirstSet.Or(FollowSet);
    LastSet.Or(PrecedeSet);
    END;
  END ComputeAttachedSets;
```

The production set is slightly different from the other sets listed in these chunks, since it is made of non-terminals (See class *DuraFlat.NonTerminalSet* on page 560) while $FIRST$ and $FOLLOW$ are made of terminals (See class *DuraFlat.TerminalSet* on page 558).

The production set P_n for a non-terminal n is defined as $x \in P_n \Leftrightarrow n \stackrel{*}{\Rightarrow} x$.

507b ⟨*Computing the rule's production set, definition* 507b⟩≡

```
METHOD ComputeProductionSet;
METHOD ProductionSet: NonTerminalSet;
METHOD SetProductionSet(TheSet: NonTerminalSet);
```

A non-terminal's production set is the union of its right hand sides production sets (See method *DuraFlat.FlatRHS.AccumulateProductionSet* on page 537).

508 ⟨*Computing the rule's production set, implementation* 508⟩≡

```
VAR
  TheProductionSet: NonTerminalSet;

METHOD ProductionSet: NonTerminalSet;
  BEGIN
  RESULT ← TheProductionSet;
  END ProductionSet;

METHOD SetProductionSet(TheSet: NonTerminalSet);
  BEGIN
  TheProductionSet ← TheSet;
  END SetProductionSet;

METHOD ComputeProductionSet;
  BEGIN
  ASSERT ProductionSet ≠ Λ;
  FOR RHS ∈ THIS DO
    RHS.AccumulateProductionSet (ProductionSet);
    END;
  END ComputeProductionSet;
```

7.13.5 Comparing FlatRules

We would like to be able to compare two `FlatRule` instances for equality, so that we can merge rules if they happen to be duplicated in several contexts. For instance, when dealing with `COBOL` , where the "IS" keyword is often acceptable but almost always optional, a rule such as:

$$\begin{array}{lll} N_{IS} & ::= & \text{“}IS\text{''} \mid \\ & & \epsilon \end{array}$$

occurs a dozen times.

The `Compare` method listed hereunder compares two `FlatRule` instances, so that lists of `FlatRule` can be sorted, and equivalent rules can be merged.

`AlmostEqual` qualifies a situation where two rules cannot be distinguished in terms of the order relationship (See `Compare` method below), but where they are not truly `Equal` in the sense that they cannot be merged. This will typically be the case when two grammar rules are strictly equivalent, except for their attached reduction.

Sorting a list of `FlatRule` instances using the comparison method described here will yield equivalence classes, where two elements in such an equivalence class are `Equal` or `AlmostEqual`.

509a ⟨*Comparing with another* `FlatRule`*, definition* 509a⟩≡

```
CONST
  Smaller = -1;
  Equal = 0;
  AlmostEqual = 1;
  Greater = 2;

METHOD Compare (Other: FlatRule): INTEGER;
```

Comparing `FlatRules` is based on the comparison of `FlatRHS` (See method *DuraFlat.FlatRHS.Compare* on page 532) . . .

509b ⟨*Comparing with another* `FlatRule`*, implementation* 509b⟩≡

```
METHOD Compare (Other: FlatRule): INTEGER;
  VAR
    j: INTEGER;
  BEGIN
  IF Size > Other.Size THEN
    RESULT ← Greater;
   ELSIF Size < Other.Size THEN
    RESULT ← Smaller;
   ELSE
    FOR i ∈ 0 .. Size-1
        WHILE (RESULT = Equal) ∨ (RESULT = AlmostEqual) DO
      j ← THIS_i.Compare (Other_i);
      IF j ≠ Equal THEN
        RESULT ← j;
        END;
      END;
    END;
  END Compare;
```

7.13.6 Producing `yacc` equivalent

Mainly for research and comparison purposes, `DURA` provides the ability to generate a `yacc` grammar source file.

509c ⟨*Generating equivalent* `yacc` *source, definition* 509c⟩≡

```
METHOD GenerateYacc (Output: OutputStream);
```

Generating a `yacc` grammar rule for a `DURA` grammar rule is just a question of producing a disjunction of the right hand sides it contains (See page 535). `GenerateYacc` also generates grammar rules for token class transitions (See method *DuraFlat.FlatRHS.GenerateYaccTokenClasses* on page 536).

510a ⟨*Generating equivalent* `yacc` *source, implementation* 510a⟩≡

```
METHOD GenerateYacc (Output: OutputStream);
  BEGIN
  GenerateYaccIdent (Output);
  Output.WriteLine (":");
  FOR i ∈ 0 .. Size - 1 DO
    IF i > 0 THEN
      Output.WriteLine (" |");
      END;
    Get(i).GenerateYacc (Output);
    END;
  Output.WriteLine(" ;");
  Output.WriteLn;
  FOR Rhs ∈ THIS DO
    Rhs.GenerateYaccTokenClasses (Output);
    END;
  END GenerateYacc;
```

⟨`GenerateYaccIdent` *method implementation* 510b⟩

`GenerateYaccIdent` produces a valid identifier that describes the rule, using its attached class name if relevant, or a synthetic identifier based on the rule number. In any case, dots are replaced by underscore characters to produce valid identifiers.

510b ⟨`GenerateYaccIdent` *method implementation* 510b⟩≡

```
METHOD GenerateYaccIdent (Output: OutputStream);
  VAR
    a: ARRAY OF CHAR;
  BEGIN
  IF Class ≠ Λ THEN
    a ← Class.DotImage.CLONE;
   ELSE
    a ← "R"+ IntConversions.IntToString (Nr, 0);
    END;
  FOR i ∈ 0 .. a.SIZE - 1 | a_i = "." DO
    a_i ← "_";
    END;
  Output.WriteString (a);
  END GenerateYaccIdent;
```

7.13.7 Right hand sides

`FlatRHS` implements right hand sides of grammar rules.

The ObjectZ definition of `FlatRHS` will require a little utility function defined by induction:

A `FlatRHS` can then be defined as follows:

FlatRHS

Components is implemented using implementation inheritance. In other words, there is no such thing as a *Components* attribute in the actual implementation of this class: this service is provided by inheriting from *ArrayList*, even if this does not reflect a true inheritance relationship in terms of subtyping and polymorphism.

$Components : \mathrm{seq} \downarrow FlatRuleComponent$ (See page 540)
$Length : \mathbb{N}$
$Enclosing : \rho FlatRule$ (See page 477)
$Items : \mathrm{seq}\, FlatItem$ (See page 623)
$FirstItem, LastItem : \rho FlatItem$

$Length = \# Components$
$\# Items = Length + 1$

The following constraint defines the relationship between the *FlatRHS* and the *FlatItem*s and *FlatRuleComponent*s it contains. The relationship between the *FlatRHS* and its enclosing *FlatRule* is described with the latter (See page 477).

$\forall\, i \in Items \bullet \delta i.Enclosing = self$
$\forall\, c \in Components \bullet \delta c.Enclosing = self$
$self \in \delta Enclosing.RHS$

The *first* and *last* operators, as defined by Z can only be applied to non-empty lists. This is guaranteed here by the relationship between the lengths of *Items* and *Component* (see above) which implies that *Items* cannot be an empty sequence.

$\delta FirstItem = first\ Items$
$\delta LastItem = last\ Items$

The production set attached to a $FlatRHS$ is a set of *references* to flat rules (hence the ρ). See page 979 for more information about ObjectZ references as used in this document.

$ProductionSet : \mathbb{P}\,\rho FlatRule$

Since p is not a sequence, hence, not a valid $SymbolString$ (See page 476), we must use a sequence made of the a single element $\langle p \rangle$ to test membership in $Strings$.

$p \in ProductionSet \Leftrightarrow \langle p \rangle \in Strings$

$NullableLevel : NullableLevelType$ (See page 476)

$$
\begin{array}{l}
NullableLevel = AlwaysNull \Leftrightarrow \\
\quad \forall\, c \in \operatorname{ran} Components \bullet c.NullableLevel = AlwaysNull \\
NullableLevel = Nullable \Leftrightarrow \\
\quad \exists\, c \in Components \bullet c.NullableLevel \neq AlwaysNull \land \\
\qquad \forall\, c \in Components \bullet c.NullableLevel \neq NotNullable
\end{array}
$$

$TerminalStrings$ is simply the restriction of $Strings$ to complete derivations only, that is, derivations without non-terminal references.
Given a sequence s, the distributed concatenation $\frown/s$ is the concatenation of all elements in s, that is, $s(1)\frown s(2)\frown s(3)\frown \ldots$

$Strings, TerminalStrings : \mathbb{P}\,SymbolString$

$$
\begin{array}{l}
s \in Strings \Leftrightarrow \\
\quad \exists\, Words \in \operatorname{seq} SymbolString \bullet \\
\qquad \#Words = \#Components \land \\
\qquad\quad s = \frown/\,Words \land \\
\qquad\qquad \forall\, i \in \operatorname{dom} Words \bullet Words(i) \in Components(i).Strings \\
TerminalStrings = \{x \in Strings \mid \operatorname{ran} x \subset TokenCode\}
\end{array}
$$

$MiddleSet$ holds the set of terminals (See page 14) that can occur inside (understand: not as leading or trailing symbol) the derivations of a $FlatRHS$.

$MiddleSet : \mathbb{P}\,TokenCode$

$$MiddleSet = \bigcup_{x \in TerminalStrings} \operatorname{ran} front\ tail\ x$$

$FirstSet$ returns the set of terminals that can *start* the derivations of a $FlatRHS$.

$$FirstSet : \mathbb{P}\ TokenCode$$

$$FirstSet = \bigcup_{x \in TerminalStrings} \{head\ x\}$$

$LastSet$ returns the set of terminals that can *end* the derivations of a $FlatRHS$.

$$LastSet : \mathbb{P}\ TokenCode$$

$$LastSet = \bigcup_{x \in TerminalStrings} \{last\ x\}$$

See also figure 7.8 on page 620 for more information about how `FlatRHS`, `FlatItem` and `FlatRuleComponents` interact.

`FlatRHS` inherits from a list of `FlatRuleComponent` (See page 540) so that one can use quantifiers on a right hand side to visit all its components conveniently.

513 ⟨`FlatRHS` *class definition* 513⟩≡

```
CLASS FlatRHS;
  INHERITS ArrayList(FlatRuleComponent);

  ⟨Attaching a reduction to a rule, definition 519b⟩
  ⟨Checking a FlatRHS's nullability, definition 515a⟩
  ⟨Checking a FlatRHS's reduceability, definition 516a⟩
  ⟨Dereferencing canonical RHS, definition 522a⟩
  ⟨FlatRHS's items management, definition 516c⟩
  ⟨Dumping a FlatRHS on an output stream, definition 519d⟩
  ⟨Numbering method, definition 525c⟩
  ⟨Replacing a FlatRHS component by an array of components, definition 524e⟩
  ⟨Recovery and snip handling, definition 517a⟩

  ⟨Attaching a FlatRHS to its enclosing FlatRule, definition 524c⟩
  ⟨Reduction code generation, definition 526b⟩
  ⟨Comparison method, definition 531c⟩

END FlatRHS;
```

A large part of `FlatRHS` class's functionality is defined by its inheritance relationship to `ArrayList`.

However, it also provides additional features, such a textual representation by means of the `Dump` method, the `Nullable` method, and others.

514 ⟨`FlatRHS` *class implementation* 514⟩≡

```
CLASS FlatRHS;
  INHERITS ArrayList(FlatRuleComponent);

  ⟨Checking a FlatRHS's nullability, implementation 515b⟩
  ⟨Checking a FlatRHS's reduceability, implementation 516b⟩
  ⟨FlatRHS's items management, implementation 516d⟩

  ⟨FlatRHS closure management 519a⟩
  ⟨Attaching a reduction to a rule, implementation 519c⟩
  ⟨Cloning a FlatRHS 521b⟩

  ⟨Dereferencing canonical RHS, implementation 522b⟩

  ⟨Dumping a FlatRHS on an output stream, implementation 520⟩

  ⟨Replacing a FlatRHS component by an array of components, implementation 525a⟩

  ⟨Getting the original grammar rule that generated a given FlatRHS 523a⟩
  ⟨Consulting a FlatRHS precedence 523b⟩
  ⟨Consulting a FlatRHS associativity 524a⟩
  ⟨Recovery and snip handling, implementation 518a⟩

  ⟨Attaching a FlatRHS to its enclosing FlatRule, implementation 524d⟩

  ⟨Numbering method, implementation 526a⟩
  ⟨Reduction code generation, implementation 526c⟩
  ⟨Visiting a FlatRHS to compute its attached sets (FIRST,FOLLOW) 528b⟩

  ⟨Redefining list operations on FlatRHS 530b⟩
  ⟨Comparison method, implementation 532a⟩

  ⟨Generating the equivalent yacc source, implementation 536a⟩

  ⟨Middle set management 535a⟩

  ⟨Equivalence class management, implementation 530a⟩
  ⟨Partial production set computation 537⟩

END FlatRHS;
```

Deciding whether a given right hand side is nullable or not is a pretty straightforward operation: we basically defer this responsibility to the terminals (which are never nullable) and non-terminals (which can be nullable or not) that compose `THIS` (See method *DuraFlat.FlatRuleComponent.NullableLevel* on page 540).

515a ⟨*Checking a* `FlatRHS`*'s nullability, definition* 515a⟩≡
```
METHOD Nullable: BOOLEAN;
METHOD NullableLevel: INTEGER;
```

515b ⟨*Checking a* `FlatRHS`*'s nullability, implementation* 515b⟩≡
```
METHOD NullableLevel: INTEGER;
  VAR
    Min, Max, k: INTEGER;
  BEGIN
  IF Size = 0 THEN
    RESULT ← FlatRule.CodeAlwaysNullLevel;
   ELSE
    Min ← 999;
    Max ← -999;
    FOR El ∈ THIS WHILE Max < FlatRule.CodeNotNullableLevel DO
      k ← El.NullableLevel;
      Min ← SYSTEM.MIN (Min, k);
      Max ← SYSTEM.MAX (Max, k);
      END;
    IF Max = FlatRule.CodeNotNullableLevel THEN
      RESULT ← FlatRule.CodeNotNullableLevel;
     ELSIF Min = FlatRule.CodeUnknownLevel THEN
      RESULT ← FlatRule.CodeUnknownLevel;
     ELSE
      RESULT ← Max;
      END;
    END;
  END NullableLevel;
```

515c ⟨*Checking a* `FlatRHS`*'s nullability, implementation* 515b⟩+≡
```
METHOD Nullable: BOOLEAN;
  BEGIN
  RESULT ← NullableLevel ≠ FlatRule.CodeNotNullableLevel;
  END Nullable;
```

Checking whether a given right-hand side always produces the empty string is pretty straightforward as well . . .

515d ⟨*Checking a* `FlatRHS`*'s nullability, implementation* 515b⟩+≡
```
METHOD AlwaysNull: BOOLEAN;
  BEGIN
  RESULT ← ∀ c ∈ THIS • c.AlwaysNull;
  END AlwaysNull;
```

516a ⟨*Checking a* `FlatRHS`*'s reduceability, definition* 516a⟩≡
```
METHOD Reducible: BOOLEAN;
```

A right-hand side is reducible if all its elements are reducible (See method *DuraFlat.FlatRuleComponent.Reducible* on page 543).

516b ⟨*Checking a* `FlatRHS`*'s reduceability, implementation* 516b⟩≡
```
METHOD Reducible: BOOLEAN;
  BEGIN
  RESULT ← ∀ c ∈ THIS • c.Reducible;
  END Reducible;
```

`GetItem` can be used to query a `FlatRHS`'s item (See figure 7.8 on page 620 to see exactly what items are all about).

516c ⟨`FlatRHS`*'s items management, definition* 516c⟩≡
```
METHOD GetItem (Pos: INTEGER): FlatItem;
```

On the first invocation, `GetItem` creates `TheItems`, and initializes it.

516d ⟨`FlatRHS`*'s items management, implementation* 516d⟩≡
```
VAR
  TheItems: ARRAY OF FlatItem;

METHOD GetItem (Pos: INTEGER): FlatItem;
  BEGIN
  IF TheItems = Λ THEN
    TheItems.CREATE (Size + 1);
    FOR i ∈ 0 .. TheItems.SIZE - 1 DO
      TheItems_i.CREATE (THIS, i);
      END;
    END;
  ASSERT Pos < TheItems.SIZE;
  ASSERT Pos ≥ 0;
  RESULT ← TheItems_Pos;
  END GetItem;
```

517a ⟨*Recovery and snip handling, definition* 517a⟩≡

```
CONST
  NoRecovery =        DuraClassDefinition.NoRecovery;
  DefaultRecovery =   DuraClassDefinition.DefaultRecovery;
  AutomaticRecovery = DuraClassDefinition.AutomaticRecovery;

METHOD RecoverSet: TerminalSet;
METHOD RecoveryLevel: INTEGER;
  POST
    NO_CHANGE;

METHOD SetRecoveryLevel(Level: INTEGER);
  PRE
    (Level = NoRecovery) ∨
    (Level = DefaultRecovery) ∨
    (Level = AutomaticRecovery);
```

517b ⟨*Recovery and snip handling, definition* 517a⟩+≡

```
CONST
  FullCut =    DuraClassDefinition.FullCut;
  DefaultCut = DuraClassDefinition.DefaultCut;
  NoCut =      DuraClassDefinition.NoCut;

METHOD CutLevel: INTEGER;
  POST
   NO_CHANGE;

METHOD SetCutLevel(Level: INTEGER);
  PRE
    (Level = FullCut) ∨
    (Level = DefaultCut) ∨
    (Level = NoCut);
```

`RecoverSet` returns a set of terminals (See class *DuraFlat.TerminalSet* on page 558) on which a right-hand side can be reduced when performing error recovery. In other words, if an error occurs while a non-terminal p is being analyzed, one of p's right hand sides can be forced to reduce when encountering one of p's recovery set terminals.

There could be various strategies regarding how to build this recovery set. In its current incarnation, `DURA` only supports *automatic error recovery*, which is based on using the non-terminal's $FOLLOW$ set for recovery. If not recovery strategy was defined, `RecoverSet` returns `VOID`.

518a ⟨*Recovery and snip handling, implementation* 518a⟩≡

```
METHOD RecoverSet: TerminalSet;
  BEGIN
  IF RecoveryLevel = AutomaticRecovery THEN
    RESULT ← GetFlatRule.FollowSet;
    END;
  END RecoverSet;
```

518b ⟨*Recovery and snip handling, implementation* 518a⟩+≡

```
VAR
  TheRecoveryLevel: INTEGER;

METHOD RecoveryLevel: INTEGER;
  BEGIN
  RESULT ← TheRecoveryLevel;
  END RecoveryLevel;

METHOD SetRecoveryLevel(Level: INTEGER);
  BEGIN
  IF TheRecoveryLevel = 0 THEN
    TheRecoveryLevel ← Level;
    END;
  END SetRecoveryLevel;
```

518c ⟨*Recovery and snip handling, implementation* 518a⟩+≡

```
VAR
  TheCutLevel: INTEGER;

METHOD CutLevel: INTEGER;
  BEGIN
  RESULT ← TheCutLevel;
  END CutLevel;

METHOD SetCutLevel(Level: INTEGER);
  BEGIN
  IF TheCutLevel = 0 THEN
    TheCutLevel ← CutLevel;
    END;
  END SetCutLevel;
```

Just as `ComputeClosure` for `FlatRules` (See page 491), `ComputeClosure` for `FlatRHS` returns a number of bits that is used to control the algorithm, as shown in `ComputeClosures` (See method *DuraFlSpec.FlatSpecification.ComputeClosures* on page 567).

The closures are computed at the item level (See method *DuraItem.FlatItem.ComputeClosure* on page 627). YAFL

Unlike most YAFL loops, it goes from 0 to `Size` rather than `Size-1`, because it visits *items*, and a RHS with n components defines $n+1$ items.

519a ⟨`FlatRHS` *closure management* 519a⟩≡

```
METHOD ComputeClosure: INTEGER;
  BEGIN
  FOR i ∈ 0 .. Size DO
    RESULT ← RESULT + GetItem(i).ComputeClosure;
    END;
  END ComputeClosure;
```

519b ⟨*Attaching a reduction to a rule, definition* 519b⟩≡

```
METHOD SetReduction (Reduction: DuraReduction);
METHOD Reduction: DuraReduction;
```

This chunk manages the `DuraReduction` (See page 822) attached to the RHS. This reduction controls the action that must be taken when the RHS is reduced (create a new object, fetch attributes, etc ...).

519c ⟨*Attaching a reduction to a rule, implementation* 519c⟩≡

```
VAR
  TheReduction: DuraReduction;

METHOD SetReduction (Reduction: DuraReduction);
  BEGIN
  TheReduction ← Reduction;
  END SetReduction;

METHOD Reduction: DuraReduction;
  BEGIN
  RESULT ← TheReduction;
  END Reduction;
```

519d ⟨*Dumping a* `FlatRHS` *on an output stream, definition* 519d⟩≡

```
METHOD Dump (Output: OutputStream);
METHOD DumpItem (Output: OutputStream;
                 Position: INTEGER);
```

520 ⟨*Dumping a* `FlatRHS` *on an output stream, implementation* 520⟩≡

```
METHOD Dump (Output: OutputStream);
  BEGIN
  Output.WriteInt (Nr, 0);
  Output.WriteChar (“/”);
  FOR i ∈ 0 .. Size - 1 DO
    IF TheItems ≠ Λ THEN
      GetItem(i).Dump(Output);
      Output.WriteChar (“ ”);
      END;
    Get(i).Dump (Output);
    Output.WriteChar (“ ”);
    END;
  IF TheItems ≠ Λ THEN
    GetItem(Size).Dump(Output);
    END;
  IF Nullable THEN
    Output.WriteString (“<NULLABLE> ”);
   ELSIF AlwaysNull THEN
    Output.WriteString (“<NULL> ”);
   ELSE
    Output.WriteString (“<NOTNULL> ”);
    END;
  IF CutLevel = FullCut THEN
    Output.WriteString (“<SNIP> ”);
    END;
  IF RecoveryLevel = AutomaticRecovery THEN
    Output.WriteString (“<RECOVER> ”);
    END;
  IF Reduction ≠ Λ THEN
    Reduction.Dump (Output);
    END;
  Output.WriteLn;
  END Dump;
```

`DumpItem` outputs a right-hand side together with a position marked by producing a “*”.

521a ⟨*Dumping a* `FlatRHS` *on an output stream, implementation* 520⟩+≡

```
METHOD DumpItem (Output: OutputStream;
                 Position: INTEGER);
  BEGIN
  IF GetFlatRule ≠ Λ THEN
    Output.WriteInt (GetFlatRule.Nr, 0);
    Output.WriteChar (“:”);
    END;
  Output.WriteInt (Nr, 0);
  Output.WriteChar (“:”);
  FOR i ∈ 0 .. Size - 1 DO
    IF i = Position THEN
      Output.WriteString (“* ”);
      END;
    Get(i).Dump (Output);
    Output.WriteChar (“ ”);
    END;
  IF Position = Size THEN
    Output.WriteChar (“*”);
    END;
  Output.WriteLn;
  END DumpItem;
```

`CLONE`ing a right-hand side means attaching the newly created clone to its origin, so that every operation (insertion, delete) performed on any element of the so-defined equivalence class is automatically performed to all the elements of the equivalence class.

This wicked scheme allows us to paste `CLONE`d versions of a rule in several places, guaranteeing us that any change we perform afterwards on any of the versions of the rule will be performed on all its versions, thereby ensuring that the duplication process does not jeopardize the system’s consistency (See method *DuraFlat.FlatRHS.AttachToEquivalenceClass* on page 530).

521b ⟨*Cloning a* `FlatRHS` 521b⟩≡

```
REDEFINE METHOD CLONE (Other: FlatRHS);
  BEGIN
  TheItems ← Λ;
  Next ← Λ;
  Previous ← Λ;
  AttachToEquivalenceClass (Other);
  END CLONE;
```

DURA generates quite a few grammar rules automatically. Hence, many phony intermediate non-terminals are created, and the purpose of the Canonical is to abstract these phony non-terminals, and to return the "significant" right-hand side corresponding to THIS.

THIS is only a chain to a more significant RHS if three conditions are met:

- THIS only includes a single FlatRuleComponent.
- This single FlatRuleComponent is a FlatNonTerminal.
- The grammar rule attached to this FlatNonTerminal has a single right hand side.

If all these conditions are fulfilled, then Canonical is called recursively on this single FlatRHS that has been found. Otherwise, THIS is returned.

522a ⟨*Dereferencing canonical RHS, definition* 522a⟩≡

```
METHOD Canonical: FlatRHS;
```

522b ⟨*Dereferencing canonical RHS, implementation* 522b⟩≡

```
METHOD Canonical: FlatRHS;
  BEGIN
  ⟨Set the return value to THIS 522c⟩
  IF Size = 1 THEN
    WHAT First OF
      IN FlatNonTerminal:
        IF TAG.Rule.Size = 1 THEN
          RESULT ← TAG.Rule.First.Canonical;
          ASSERT RESULT ≠ Λ;
          END;
        END;
     ELSE
      -- Don't abort...
      END;
    END;
  END Canonical;
```

As explained above, if no chain can be followed to a canonical right hand side, then the FlatRHS is its own canonical form.

522c ⟨*Set the return value to* THIS 522c⟩≡

```
RESULT ← THIS;
```

A right hand side can have a precedence, that matches the precedence of the grammar rule that generated it. In practice, the precedence is returned by the attached reduction if it is a FetchNonTerminalReduction (See page 825), since the reduction then holds a reference to the originating grammar rule as an InterRule (See page 742).

523a ⟨*Getting the original grammar rule that generated a given* `FlatRHS` 523a⟩≡
```
METHOD Rule: InterRule;
  BEGIN
  IF Reduction ≠ Λ THEN
    WHAT Reduction OF
      IN FetchNonTerminalReduction:
        RESULT ← TAG.Rule;
        END;
     ELSE
      -- Don't abort, default is VOID
      END;
    END;
  END Rule;
```

A RHS has a non-zero precedence if and only if it has an attached `InterRule`, in which case `Precedence` for `InterRule` can be used (See page 745).

523b ⟨*Consulting a* `FlatRHS` *precedence* 523b⟩≡
```
METHOD Precedence: INTEGER;
  VAR
    r: InterRule;
  BEGIN
  r ← Rule;
  IF r ≠ Λ THEN
    RESULT ← r.Precedence;
    END;
  END Precedence;
```

`CanonicPrecedence` is a simple extension of the method described above, that returns the precedence of the canonical right hand side attached to `THIS`.

523c ⟨*Consulting a* `FlatRHS` *precedence* 523b⟩+≡
```
METHOD CanonicPrecedence: INTEGER;
  BEGIN
  RESULT ← Canonical.Precedence;
  END CanonicPrecedence;
```

Just as for precedence (See method *DuraFlat.FlatRHS.Precedence* on the page before), associativity is only meaningful if the right-hand side denoted by `THIS` has an attached `InterRule` (See method *DuraIntermediateFormat.InterRule.Associativity* on page 743).

524a ⟨*Consulting a* `FlatRHS` *associativity* 524a⟩≡
```
METHOD Associativity: INTEGER;
   VAR
      r: InterRule;
   BEGIN
   r ← Rule;
   IF r ≠ Λ THEN
      RESULT ← r.Associativity;
      END;
   END Associativity;
```

`CanonicAssociativity` is a simple extension of the method described above, that returns the associativity of the canonical right hand side attached to `THIS`.

524b ⟨*Consulting a* `FlatRHS` *associativity* 524a⟩+≡
```
METHOD CanonicAssociativity: INTEGER;
   BEGIN
   RESULT ← Canonical.Associativity;
   END CanonicAssociativity;
```

524c ⟨*Attaching a* `FlatRHS` *to its enclosing* `FlatRule`*, definition* 524c⟩≡
```
METHOD GetFlatRule: FlatRule;
   POST
      NO_CHANGE;
```

524d ⟨*Attaching a* `FlatRHS` *to its enclosing* `FlatRule`*, implementation* 524d⟩≡
```
VAR
   TheFlatRule: FlatRule;

METHOD AttachToFlatRule (Rule: FlatRule);
   BEGIN
   TheFlatRule ← Rule;
   END AttachToFlatRule;

METHOD GetFlatRule: FlatRule;
   BEGIN
   RESULT ← TheFlatRule;
   END GetFlatRule;
```

`ReplaceNonTerminal` replaces an element from a `FlatRHS` at index `Position` by the elements found in `RHS`. This facility is used by the `Normalize` method (See page 568).

524e ⟨*Replacing a* `FlatRHS` *component by an array of components, definition* 524e⟩≡
```
METHOD ReplaceComponent (Position: INTEGER;
                         RHS: FlatRHS);
```

525a ⟨*Replacing a* `FlatRHS` *component by an array of components, implementation* 525a⟩≡

```
METHOD ReplaceComponent (Position: INTEGER;
                         RHS: FlatRHS);
  VAR
    Target: InterAttribute;
  BEGIN
  ASSERT TheItems = Λ;
  ASSERT Position ≥ 0;
  ASSERT Position < Size;
  Target ← THIS_Position.Target;
  Delete (Position);
  FOR i ∈ 0 .. RHS.Size - 1 DO
    Insert (Position + i, RHS_i);
    END;
  IF Target ≠ Λ THEN
    ⟨Set target 525b⟩
    END;
  END ReplaceComponent;
```

If there was a target on the position we replace, we must restore this target in the pasted RHS. This operation is of course only meaningful if the right-hand side to paste is made of a single component. If it is made of more than one component, the replacement of a component with a target should not be permitted.

525b ⟨*Set target* 525b⟩≡

```
ASSERT RHS.Size = 1;
ASSERT THIS_Position.Target = Λ;
THIS_Position.SetTarget (Target);
```

In order to be able to represent grammars in tables, we need the ability to number grammar rules, states, RHS's, etc. The following chunks implement this service for RHS's.

525c ⟨*Numbering method, definition* 525c⟩≡

```
METHOD Nr: INTEGER;
METHOD SetNr (Nr: INTEGER);
```

526a ⟨*Numbering method, implementation* 526a⟩≡

```
VAR
  TheNr: INTEGER;

METHOD Nr: INTEGER;
  BEGIN
  RESULT ← TheNr;
  END Nr;

METHOD SetNr (Nr: INTEGER);
  BEGIN
  TheNr ← Nr;
  END SetNr;
```

7.13.8 Code generation

526b ⟨*Reduction code generation, definition* 526b⟩≡

```
METHOD RequiresAdHocReduction: BOOLEAN;
METHOD GenerateReduceMethodCall (Output: DuraOutputStream);
METHOD GenerateReduceMethodBody (Output: DuraOutputStream);
```

`RequiresAdHocReduction` indicates whether a right-hand side requires a specific method for its reduction. Such a method is required if at least one of two conditions is fulfilled:

- We generate a compiled parsers. Interpreted parsers have a generic reduction scheme, so that they only require ad hoc reductions for special cases (see below)
- A reduction is attached to the RHS, and `Implicit` (See page 823) returns `FALSE`, indicating that the reduction is not covered by the generic reduction scheme. For instance, a reduction that extracts parts of the LR stack onto attributes cannot be ignored, while a reduction that implements reverse inheritance basically does nothing, and can be ignored safely.

526c ⟨*Reduction code generation, implementation* 526c⟩≡

```
METHOD RequiresAdHocReduction: BOOLEAN;
  BEGIN
  RESULT ← ¬ DuraCfg.GenerateInterpretedParsers ∨
                ((Reduction ≠ Λ) ∧ ¬ Reduction.Implicit);
  END RequiresAdHocReduction;
```

`ReduceMethodName` just returns a unique method name for the reduction of `THIS`, based on the numbering scheme.

526d ⟨*Reduction code generation, implementation* 526c⟩+≡

```
METHOD ReduceMethodName: ARRAY OF CHAR;
  BEGIN
  RESULT ← "Reduce_" + IntConversions.IntToString (Nr, 0);
  END ReduceMethodName;
```

The code generation methods listed below take a `DuraOutputStream` as parameter (See page 859), which is an `OutputStream` enriched to support the convenient generation of YAFL source files. Among other facilities, it takes care of indentation automatically.

Calling this reduction method becomes rather simple . . .

527a ⟨*Reduction code generation, implementation* 526c⟩+≡

```
METHOD GenerateReduceMethodCall (Output: DuraOutputStream);
  BEGIN
  Output.WriteLine (ReduceMethodName + ‘‘;”);
  END GenerateReduceMethodCall;
```

while producing its implementation is slightly more complex.

527b ⟨*Reduction code generation, implementation* 526c⟩+≡

```
METHOD GenerateReduceMethodBody (Output: DuraOutputStream);
  VAR
    Name: ARRAY OF CHAR;
  BEGIN
  IF RequiresAdHocReduction THEN
    Name ← ReduceMethodName;
    Output.WriteLine (‘‘METHOD ” + Name + ‘‘;”);
    Output.IncIndent (2);
    IF (Reduction ≠ Λ) ∧ ¬ Reduction.Implicit THEN
      ⟨Ask the attached reduction to generate the code 527c⟩
     ELSE
      ⟨Generate a generic reduction 528a⟩
      END;
    Output.WriteLine (‘‘END ” + Name + ‘‘;”);
    Output.DecIndent (2);
    Output.WriteLn;
    END;
  END GenerateReduceMethodBody;
```

If the attached reduction is not trivial (See method *DuraReductions.DuraReduction.Implicit* on page 823), it takes care of its own reduction code generation scheme (See method *DuraReductions.DuraReduction.GenerateReductionCode* on page 823).

527c ⟨*Ask the attached reduction to generate the code* 527c⟩≡

```
Output.WriteString (‘‘-- ”);
Reduction.Dump (Output);
Output.WriteLn;
Reduction.GenerateReductionCode (Output, THIS);
```

The generic reduction is just a matter of calling `ReduceTopSlice` (See page 877) with, as first, parameter, the number of element to reduce from the LR stack. This number is returned by `Size`, that is, the number of `FlatRuleComponents` that are found in `THIS`. The new state the automaton moves to is computed dynamically by calling the goto method attached to the enclosing `FlatRule` (See method *DuraFlat. FlatRule.GenerateGotoMethodCall* on page 501).

528a ⟨*Generate a generic reduction* 528a⟩≡

```
Output.WriteLine (“BEGIN”);
Output.WriteString(“VOID := ReduceTopSlice (”);
Output.WriteInt (Size, 0);
Output.WriteString (“, ”);
Output.WriteInt (Nr, 0);
Output.WriteString (“, ”);
GetFlatRule.GenerateGotoMethodCall(Output, Size);
Output.WriteLine (“);”);
```

Just as nullability and reduceability for `FlatRule`'s, computing the *FIRST* and *FOLLOW* sets for a `FlatRHS` is a global issue, handled at a global level (See method *DuraFlSpec.FlatSpecification.ComputeAttachedSets* on page 602). `ComputeAttached-Sets` as defined here computes a *generation* of these sets for a single `FlatRHS`, given what is known at this stage. This method might be called repeatedly as the *FIRST* and *FOLLOW* sets of the various rules evolve.

528b ⟨*Visiting a* `FlatRHS` *to compute its attached sets (*FIRST,FOLLOW*)* 528b⟩≡

```
METHOD ComputeAttachedSets;
  BEGIN
  FOR i ∈ 0 .. Size - 1 DO
    WHAT Get(i) OF
      IN FlatNonTerminal:
        IF i < Size - 1 THEN
          Get(i+1).AddToFollowSet(TAG.Rule);
         ELSIF i > 0 THEN
          Get(i-1).AddToPrecedeSet(TAG.Rule);
          END;
        END;
     ELSE
      END;
    END;
  ⟨If the last component is a nonterminal, its FOLLOW includes mine 529a⟩
  ⟨If the first component is a nonterminal, its PRECEDE includes mine 529b⟩
  END ComputeAttachedSets;
```

529a ⟨*If the last component is a nonterminal, its* FOLLOW *includes mine* 529a⟩≡

```
WHAT Last OF
  IN FlatNonTerminal:
    TAG.Rule.FollowSet.Or(GetFlatRule.FollowSet);
    END;
 ELSE
  END;
```

529b ⟨*If the first component is a nonterminal, its* PRECEDE *includes mine* 529b⟩≡

```
WHAT First OF
  IN FlatNonTerminal:
    TAG.Rule.PrecedeSet.Or(GetFlatRule.PrecedeSet);
    END;
 ELSE
  END;
```

The following chunk handles *equivalence classes* for RHS. Whenever a list operation (insertion or deletion of a component) is performed on one element of the equivalent class, it is automatically performed on all other elements of the equivalence class as well. `AttachToEquivalenceClass` is used when `CLONE`ing a RHS (See page 521).

Busy serves as guard to indicate that a list operation is currently being performed on THIS. For instance, when appending an element to THIS, before calling Append applied on its attached next FlatRHS instance, the Busy flag will be set to ensure that we don't end up with an infinite recursion when the next FlatRHS will, in turn, apply Append on its own attached Next FlatRHS instance.

530a ⟨*Equivalence class management, implementation* 530a⟩≡

```
VAR
  Busy: BOOLEAN;
  Next, Previous: FlatRHS;

METHOD AttachToEquivalenceClass (Other: FlatRHS);
  BEGIN
  ASSERT Next = Λ;
  ASSERT Previous = Λ;
  IF Other.Next = Λ THEN
    ASSERT Other.Previous = Λ;
    Next ← Other;
    Previous ← Other;
    Other.Next ← THIS;
    Other.Previous ← THIS;
   ELSE
    Next ← Other.Next;
    Previous ← Other.Previous;
    Other.Next ← THIS;
    Next.Previous ← THIS;
    END;
  END AttachToEquivalenceClass;
```

Redefining these list operations serves several purposes:

- Ensure that all elements appended onto a FlatRHS instance are non-VOID
- Implement the equivalence classes. Whenever an operation is performed onto an element of the equivalence class, it must be performed on the entire class as well.

530b ⟨*Redefining list operations on* FlatRHS 530b⟩≡

```
REDEFINE METHOD Append(El: FlatRuleComponent);
  BEGIN
  IF (El ≠ Λ) ∧ ¬ Busy THEN
    Busy ← TRUE;
    BASE (El);
    IF Next ≠ Λ THEN
      Next.Append (El);
      END;
    Busy ← FALSE;
    END;
  END Append;
```

531a ⟨*Redefining list operations on* `FlatRHS` 530b⟩+≡

```
REDEFINE METHOD Delete(Pos: INTEGER);
  BEGIN
  IF ¬ Busy THEN
    Busy ← TRUE;
    BASE (Pos);
    IF Next ≠ Λ THEN
      Next.Delete(Pos);
      END;
    Busy ← FALSE;
    END;
  END Delete;
```

531b ⟨*Redefining list operations on* `FlatRHS` 530b⟩+≡

```
REDEFINE METHOD Insert(Pos: INTEGER; El: FlatRuleComponent);
  BEGIN
  IF ¬ Busy THEN
    Busy ← TRUE;
    BASE (Pos, El);
    IF Next ≠ Λ THEN
      Next.Insert(Pos, El);
      END;
    Busy ← FALSE;
    END;
  END Insert;
```

7.13.9 Comparing RHS's

531c ⟨*Comparison method, definition* 531c⟩≡

```
CONST
  Smaller = FlatRule.Smaller;
  Equal   = FlatRule.Equal;
  AlmostEqual = FlatRule.AlmostEqual;
  Greater = FlatRule.Greater;

METHOD Compare (Other: FlatRHS): INTEGER;
```

We need the ability to sort lists of RHS's to have a canonic order in which the RHS's of a grammar rule will be stored, possibly for comparison with another grammar rule. On the other hand, the actual order relationship it implements is of little actual importance, as long as it satisfies all the requirement for order relationships. Therefore, when comparing two RHS's, we first compare the "easy" things, such as length, cut level and recovery level. It is only if these three comparison did not show any difference that we actually compare the content of the RHS.

532a ⟨*Comparison method, implementation* 532a⟩≡

```
METHOD Compare (Other: FlatRHS): INTEGER;
  VAR
    j: INTEGER;
    RecursiveOk,
    ThisRecurs,
    OtherRecurs: BOOLEAN;
  BEGIN
  ⟨First, test sizes 532b⟩
  ⟨Then, test cut levels 532c⟩
  ⟨Try recovery levels ... 532d⟩
   ELSE
    ⟨Ok, now test the content ! 533⟩
    END;
  END Compare;
```

532b ⟨*First, test sizes* 532b⟩≡

```
IF (Size > Other.Size) THEN
  RESULT ← Greater;
 ELSIF (Size < Other.Size) THEN
  RESULT ← Smaller;
```

532c ⟨*Then, test cut levels* 532c⟩≡

```
 ELSIF (CutLevel > Other.CutLevel) THEN
  RESULT ← Greater;
 ELSIF (CutLevel < Other.CutLevel) THEN
  RESULT ← Smaller;
```

532d ⟨*Try recovery levels ...* 532d⟩≡

```
 ELSIF (RecoveryLevel > Other.RecoveryLevel) THEN
  RESULT ← Greater;
 ELSIF (RecoveryLevel < Other.RecoveryLevel) THEN
  RESULT ← Smaller;
```

533 ⟨*Ok, now test the content !* 533⟩≡

```
ASSERT Size = Other.Size;
ASSERT CutLevel = Other.CutLevel;
ASSERT RecoveryLevel = Other.RecoveryLevel;

RecursiveOk ← TRUE;
RESULT ← Equal;

FOR i ∈ 0 .. Size-1 WHILE RecursiveOk DO
  j ← THIS_i;
  IF j ≠ Equal THEN
    ⟨Check whether the two rules are recursive in the same way 534a⟩
    END;
  IF (RESULT = Equal) ∨
      ((RESULT = AlmostEqual) ∧
      ((j = Greater) ∨ (j = Smaller))) THEN
    RESULT ← j;
    END;
  END;
IF RecursiveOk THEN
  RESULT ← Equal;
  END;
IF RESULT = Equal THEN
  ⟨Check the reductions 534b⟩
  END;
```

This chunk detects a specific case of equality which deals with recursive rules. For instance, if I have two rules A and B defined as:

$$\begin{array}{lll} A & ::= & A\,t \mid \\ & & t \\ B & ::= & B\,t \mid \\ & & t \end{array}$$

it is clear that A and B are equivalent, and all references to B could safely be replaced by references to A. However, unless such cases are considered explicitely, comparing the first RHS of A and B (that is, $A\ ::=\ A\,t$ and $B\ ::=\ B\,t$) would yield `Greater` or `Smaller`.

The following chunk handles this case. If, so far, comparing the two RHS did not yield any difference, but comparing the components on position i yields `Greater` or `Smaller`, we check whether the two components are not recursive references to their respective enclosing rule.

534a ⟨*Check whether the two rules are recursive in the same way* 534a⟩≡
```
WHAT THISᵢ OF
  IN FlatNonTerminal:
    ThisRecurs ← TAG.Rule = GetFlatRule;
    END;
 ELSE
  ThisRecurs ← FALSE;
  END;
WHAT Otherᵢ OF
  IN FlatNonTerminal:
    OtherRecurs ← TAG.Rule = Other.GetFlatRule;
    END;
 ELSE
  OtherRecurs ← FALSE;
  END;
IF ThisRecurs ∧ OtherRecurs THEN
  j ← Equal;
 ELSE
  RecursiveOk ← FALSE;
  IF ThisRecurs ∧ ¬ OtherRecurs THEN
    j ← Greater;
   ELSIF OtherRecurs ∧ ¬ ThisRecurs THEN
    j ← Smaller;
    END;
  END;
```

We don't want to confuse two right hand sides which attached reductions do not match exactly. This is the very purpose of the `Almost` equal value: the rules are equal in terms of their grammatical behaviours, but their reduction are not (See method *DuraReductions.DuraReduction.IsEqual* on page 824).

534b ⟨*Check the reductions* 534b⟩≡
```
IF (Reduction ≠ Λ) ∨ (Other.Reduction ≠ Λ) THEN
  IF (Reduction = Λ) ∨ (Other.Reduction = Λ) THEN
    RESULT ← AlmostEqual;
   ELSIF ¬ Reduction.IsEqual (Other.Reduction) THEN
    RESULT ← AlmostEqual;
    END;
  END;
```

535a ⟨*Middle set management* 535a⟩≡
```
METHOD ComputeMiddleSet;
  VAR
    Rule: FlatRule;
    Left, Right: INTEGER;
  BEGIN
```

Plain nullable components must be taken into account, unless they are always null, in which case they must be skipped. `Left` and `Right` give resp. the first and last index of non-always null components in `THIS`.

535b ⟨*Middle set management* 535a⟩+≡
```
Rule ← GetFlatRule;
Left ← 0;
WHILE (Left < Size) ∧ THIS_Left.AlwaysNull DO
  Left ← Left + 1;
  END;
Right ← Size - 1;
WHILE (Right > Left) ∧ THIS_Right.AlwaysNull DO
  Right ← Right - 1;
  END;
```

Then, one must scan the remaining components and add them to the middle set, truncated properly, that is, setting the parameter `LeftSkip` (resp. `RightSkip`) when we are dealing with the leftmost (resp. rightmost) element of the not always null kernel of our RHS.

535c ⟨*Middle set management* 535a⟩+≡
```
FOR i ∈ Left .. Right DO
  THIS_i.AddToMiddleSet (Rule,
                   LeftSkip ← i = Left,
                   RightSkip ← i = Right);
  END;
END ComputeMiddleSet;
```

`GenerateYacc`, as described below, produces the *body* of the `yacc` equivalent for a RHS. It is called by `GenerateYacc` (See page 509), and goes through the components it contains to produce the `yacc` rule.

Rather than using the `WHAT` control structure that weakens the YAFL type system, I should have used an ad hoc deferred method in `FlatRuleComponent`, redefined by the `FlatTerminal`, `FlatNonTerminal` and `FlatTerminalSet`. In some later release, perhaps . . .

536a ⟨*Generating the equivalent* `yacc` *source, implementation* 536a⟩≡

```
METHOD GenerateYacc (Output: OutputStream);
  BEGIN
  Output.WriteString (“   ”);
  FOR c ∈ THIS DO
    Output.WriteString (“ ”);
    WHAT c OF
      IN FlatTerminal:
        Output.WriteString (DuraCfg.LexemeImage (TAG.Code));
        END;
      IN FlatNonTerminal:
        TAG.Rule.GenerateYaccIdent (Output);
        END;
      IN FlatTerminalSet:
        TAG.GenerateYaccIdent (Output);
        END;
      END;
    END;
  END GenerateYacc;
```

`GenerateYaccTokenClasses` produces a `yacc` grammar rule for all terminal set transitions included in `THIS`, since such transitions cannot be translated into plain `yacc` transitions.

536b ⟨*Generating the equivalent* `yacc` *source, implementation* 536a⟩+≡

```
METHOD GenerateYaccTokenClasses (Output: OutputStream);
  BEGIN
  FOR c ∈ THIS DO
    WHAT c OF
      IN FlatTerminalSet:
        TAG.GenerateYacc (Output);
        END;
     ELSE
      -- If not a FlatTerminalSet, do nothing.
      END;
    END;
  END GenerateYaccTokenClasses;
```

7.13.10 Computing production sets

`AccumulateProductionSet` computes the production set of a right hand side. The production set of a right hand side RHS is defined as the set of the rules r such that $RHS \stackrel{*}{\Rightarrow} r$. Typically, when using reverse inheritance to define an $IfStatement$ class that inherits from a $Statement$ class, we have $Statement \stackrel{*}{\Rightarrow} IfStatement$

Similarly to other potentially recursive computations in `DURA`, production sets are computed by an iterative process, that recomputes the entire system until it is stable, that is, until one more round of computation does not yield any change in any rule's production set.

In order to reduce the memory requirements of this process, the method accumulates the newly found rules that are part of the production set in a set which is given as parameter, rather than creating a brand new set on each invocation, which only purpose would then be to be merged with another one.

When computing the production set of a given right hand side, one must check several things:

- If there are components that require a terminal, `THIS`'s production set is empty
- If there are more than a single non-nullable non-terminal components, `THIS`'s production set is empty.
- If there is a single non-nullable non-terminal component, `THIS`'s production set is a singleton made of that non-terminal.
- In the remaining case, that is, when all the right hand side's components refer to nullable non-terminals, then all these non-terminals are part of the production set.

537 ⟨*Partial production set computation* 537⟩≡

```
  METHOD AccumulateProductionSet(NTSet: NonTerminalSet);
    VAR
      FirstNonNullable: FlatRule;
      Applicable: BOOLEAN;

      ⟨Local method Visit 539a⟩

    BEGIN
    ⟨First, find whether there is a single non-nullable non-terminal 538a⟩
    IF Applicable THEN
      ⟨Accumulate the results 538b⟩
      END;
    END AccumulateProductionSet;
```

`Applicable` will indicate whether this RHS has a non-empty production set. For this to be the case, two conditions must be fulfilled:

- The RHS cannot contain any terminal or terminal set transition
- There can be at most a single non-nullable non-terminal transition. If there is such a non-terminal transition, it is referred to by `FirstNonNullable`.

538a ⟨*First, find whether there is a single non-nullable non-terminal* 538a⟩≡

```
Applicable ← TRUE;
FOR Comp ∈ THIS WHILE Applicable DO
  WHAT Comp OF
    IN FlatNonTerminal:
      IF ¬ TAG.Nullable THEN
        IF FirstNonNullable = Λ THEN
          FirstNonNullable ← TAG.Rule;
         ELSE
          Applicable ← FALSE;
          END;
        END;
      END;
   ELSE
    Applicable ← FALSE;
    END;
  END;
```

If we have detected that at least one rule should be added to the production set, we consider whether a single non-nullable rule has been found or not, and take appropriate action in both cases.

538b ⟨*Accumulate the results* 538b⟩≡

```
IF FirstNonNullable ≠ Λ THEN
  Visit (FirstNonNullable);
 ELSE
  FOR Component ∈ THIS DO
    WHAT Component OF
      IN FlatNonTerminal:
        Visit (TAG.Rule);
        END;
      END;
    END;
  END;
```

`Visit` adds `Rule` to the current production set, and performs a logical Or with `Rule`'s production set, since $\stackrel{*}{\Rightarrow}$ is transitive.

539a ⟨*Local method* `Visit` 539a⟩≡

```
METHOD Visit(Rule: FlatRule);
  BEGIN
  NTSet.Set(Rule.Nr);
  NTSet.Or(Rule.ProductionSet);
  END Visit;
```

Small utility class . . .

539b ⟨`RHSList` *class definition* 539b⟩≡

```
CLASS RHSList;
  INHERITS ArrayList(FlatRHS);
END RHSList;
```

YAFL

`RHSList` redefines `ShouldSwap` in such a way that sorting it will put the longest RHS first, which is the order in which we want the possible reductions in a given state to be tried.

539c ⟨`RHSList` *class implementation* 539c⟩≡

```
CLASS RHSList;
  INHERITS ArrayList(FlatRHS);

  REDEFINE METHOD ShouldSwap (One, Two: FlatRHS): BOOLEAN;
    BEGIN
    RESULT ← One.Size < Two.Size;
    END ShouldSwap;

END RHSList;
```

7.13.11 Flat rule components

`FlatRuleComponent`'s sole purpose is to serve as base class for `FlatTerminal` (See page 544), `FlatNonTerminal` (See page 554) and `FlatTerminalSet` (See page 548).

Figure 7.7 Inheritance tree for `FlatRuleComponent`

- FlatRuleComponent (540)
  - FlatTerminal (544)
  - FlatNonTerminal (554)
  - FlatTerminalSet (548)

FlatRuleComponent

$FlatRuleComponent$ is the base class for all the components that occur in a $FlatRHS$ (See page 511): terminal (See page 544), and non-terminal (See page 554) transitions. Terminal transitions are generalized by transitions on *sets of terminals* (See page 548).
$FlatRuleComponent$ just holds a number of common attributes, some of which are constrained further by the derived classes.

$Closure : \mathbb{P}\, \rho FlatItem$ (See page 623)
$NullableLevel : NullableLevelType$ (See page 476)
$Enclosing : \rho FlatRHS$ (See page 511)
$Position : \mathbb{N}$
$Strings : \mathbb{P}\, SymbolString$ (See page 476)

$Position \geq 1 \land Position \leq \# \delta Enclosing.Components$
$\delta Enclosing.Component(Position) = self$
$Closure = \delta Enclosing.Items(Position).Closure$

540 ⟨`FlatRuleComponent` *class definition* 540⟩≡

```
CLASS FlatRuleComponent;

  METHOD Target: InterAttribute;
  METHOD SetTarget (Target: InterAttribute);

  METHOD Closure: FlatItemSet;
  DEFERRED METHOD NullableLevel: INTEGER;
```

```
    METHOD Nullable: BOOLEAN;
      POST
        RESULT ⇔ NullableLevel ≠
                      FlatRule.CodeNotNullableLevel;

    METHOD AlwaysNull: BOOLEAN;

    ⟨Code management, initial definition 541a⟩
    ⟨Conveniency identification methods, definition 544a⟩

  END FlatRuleComponent;
```

Code is meant to be redefined by FlatRuleComponent-derived classes, in order to attach an integer value to each of such components for simpler sorting. Code is redefined in derived classes in such a way that $X.Code < RuleOffset \, \forall \, X \in FlatTerminal$ (See page 544), $RuleOffset \leq X.Code < TerminalSetOffset \, \forall \, X \in FlatNonTerminal$ (See page 554), $TerminalSetOffset \leq X.Code \, \forall \, X \in FlatTerminalSet$ (See page 548).

541a ⟨*Code management, initial definition* 541a⟩≡

```
  CONST
    RuleOffset = 40000;
    TerminalSetOffset = 80000;

  DEFERRED METHOD Code: INTEGER;
```

541b ⟨FlatRuleComponent *class implementation* 541b⟩≡

```
  CLASS FlatRuleComponent;

    ⟨Default Dump for a rule component 542a⟩
    ⟨Deferred set management methods 542b⟩
    ⟨Default closure for a rule component 542c⟩
    ⟨Target attribute management 542d⟩
    ⟨AlwaysNull default version 543d⟩
    ⟨Comparing rule components 543a⟩
    ⟨Checking a FlatRuleComponent's reduceability 543c⟩

    METHOD Nullable: BOOLEAN;
      BEGIN
      RESULT ← THIS.NullableLevel ≠ FlatRule.CodeNotNullableLevel;
      END Nullable;

    ⟨Conveniency identification methods, implementation 544b⟩

  END FlatRuleComponent;
```

The only generic information that can be displayed regardless of the kind of `FlatRuleComponent` is the target attribute if defined (See method *DuraFlat.FlatRuleComponent.Target* on this page).

542a ⟨*Default* `Dump` *for a rule component* 542a⟩≡

```
METHOD Dump (Output: OutputStream);
  BEGIN
  IF Target ≠ Λ THEN
    Output.WriteString (“ ” + Target.FullName + “!”);
    END;
  END Dump;
```

The following methods are called by the grammar rule to compute its $FIRST$, $LAST$, $FOLLOW$, $PRECEDE$ and $MIDDLE$ sets (See method *DuraFlat.FlatRule.ComputeAttachedSets* on page 506).

542b ⟨*Deferred set management methods* 542b⟩≡

```
DEFERRED METHOD AddToFirstSet (Rule: FlatRule);
DEFERRED METHOD AddToLastSet (Rule: FlatRule);
DEFERRED METHOD AddToFollowSet (Rule: FlatRule);
DEFERRED METHOD AddToPrecedeSet (Rule: FlatRule);
DEFERRED METHOD AddToMiddleSet (Rule: FlatRule;
                                LeftSkip,
                                RightSkip: BOOLEAN);
```

By default, a rule component has an empty closure. Terminal components will retain this default behaviour, while non-terminal component will redefine it (See method *DuraFlat.FlatNonTerminal.Closure* on page 556), and consider the closure of their attached rule.

542c ⟨*Default closure for a rule component* 542c⟩≡

```
METHOD Closure: FlatItemSet;
  BEGIN
  END Closure;
```

542d ⟨*Target attribute management* 542d⟩≡

```
VAR
  TheTarget: InterAttribute;

METHOD SetTarget (Target: InterAttribute);
  BEGIN
  TheTarget ← Target;
  END SetTarget;

METHOD Target: InterAttribute;
  BEGIN
  RESULT ← TheTarget;
  END Target;
```

543a ⟨*Comparing rule components* 543a⟩≡

```
CONST
  Smaller = FlatRule.Smaller;
  Equal   = FlatRule.Equal;
  AlmostEqual = FlatRule.AlmostEqual;
  Greater = FlatRule.Greater;
```

The default version of the comparison method depicted here is called by redefined versions, and compares the common parts among all components, namely, their ability to be attached to a target attribute.

543b ⟨*Comparing rule components* 543a⟩+≡

```
METHOD Compare(Other: FlatRuleComponent): INTEGER;
  BEGIN
  IF Target = Λ THEN
    IF Other.Target = Λ THEN
      RESULT ← Equal;
     ELSE
      RESULT ← Greater;
      END;
   ELSIF Other.Target = Λ THEN
    RESULT ← Smaller;
   ELSIF Other.Target = Target THEN
    RESULT ← Equal;
   ELSE
    RESULT ← AlmostEqual;
    END;
  END Compare;
```

`Reducible` indicates whether a `FlatRuleComponent` can be reduced, that is, whether a *finite* input can reduce to this component. For instance, any `FlatTerminal` can be reduced, when the input matches its token. On the other hand, a `FlatNonTerminal` that refers to non-terminal s for which the sole production is:

$$a \ ::= \ a\, t$$

with t being a terminal symbol cannot be reduced.

543c ⟨*Checking a* `FlatRuleComponent`*'s reduceability* 543c⟩≡

```
METHOD Reducible: BOOLEAN;
  BEGIN
  RESULT ← TRUE;
  END Reducible;
```

Unless redefined, `AlwaysNull` returns FALSE.

543d ⟨`AlwaysNull` *default version* 543d⟩≡

```
METHOD AlwaysNull: BOOLEAN;
  BEGIN
  END AlwaysNull;
```

544a ⟨*Conveniency identification methods, definition* 544a⟩≡
```
METHOD IsTerminal: BOOLEAN;
METHOD IsTerminalSet: BOOLEAN;
METHOD IsNonTerminal: BOOLEAN;
```

These are only the default implementations. These methods are redefined by derived classes to yield significant values.

544b ⟨*Conveniency identification methods, implementation* 544b⟩≡
```
METHOD IsTerminal: BOOLEAN;
  BEGIN
  END IsTerminal;

METHOD IsTerminalSet: BOOLEAN;
  BEGIN
  END IsTerminalSet;

METHOD IsNonTerminal: BOOLEAN;
  BEGIN
  END IsNonTerminal;
```

FlatTerminal

FlatRuleComponent (See page 540)

FlatTerminal is a transition to be stored in a *FlatRHS*, and which denotes a transition on token *Code*.

$Code : TokenCode$ (See page 14)

The inherited attributes are constrained in a rather predictable way. For instance, the only item it can move to without consuming characters is the item on its left, the only derivation of a *FlatTerminal* transition is a string made of *Code* and a *FlatTerminal* is therefore not nullable.

$Closure = \{\delta Enclosing.Items(Position)\}$

$NullableLevel = NotNullable$

Strings is defined as a singleton that contains a sequence made of a single element *Code*.

$Strings = \{\langle Code \rangle\}$

544c ⟨`FlatTerminal` *class definition* 544c⟩≡
```
CLASS FlatTerminal;
  INHERITS FlatRuleComponent;

  REDEFINE METHOD CREATE (Code: INTEGER);
END FlatTerminal;
```

545a ⟨`FlatTerminal` *class implementation* 545a⟩≡

```
CLASS FlatTerminal;
  INHERITS FlatRuleComponent;

  VAR
    TheCode: INTEGER;

  REDEFINE METHOD CREATE (Code: INTEGER);
    BEGIN
    BASE;
    TheCode ← Code;
    END CREATE;

  REDEFINE METHOD Dump (Output: OutputStream);
    BEGIN
    BASE (Output);
    Output.WriteString (DuraCfg.LexemeImage (TheCode));
    END Dump;
```

Of course, a terminal transition is never nullable.

545b ⟨`FlatTerminal` *class implementation* 545a⟩+≡

```
REDEFINE METHOD NullableLevel: INTEGER;
  BEGIN
  RESULT ← FlatRule.CodeNotNullableLevel;
  END NullableLevel;
```

`Code` is redefined to simply return the code of the lexeme which is of course supposed to be smaller than `RuleOffset` (See page 541).

545c ⟨`FlatTerminal` *class implementation* 545a⟩+≡

```
REDEFINE METHOD Code: INTEGER;
  BEGIN
  RESULT ← TheCode;
  ASSERT RESULT < THIS.RuleOffset;
  END Code;
```

Quite obviously, a `FlatTerminal` is a terminal transition (See method *DuraFlat. FlatRuleComponent.IsTerminal* on the facing page).

545d ⟨`FlatTerminal` *class implementation* 545a⟩+≡

```
REDEFINE METHOD IsTerminal: BOOLEAN;
  BEGIN
  RESULT ← TRUE;
  END IsTerminal;
```

Adding the `FlatTerminal` to the sets is merely a question of setting a bit here and there.

546 ⟨`FlatTerminal` *class implementation* 545a⟩+≡

```
REDEFINE METHOD AddToFirstSet (Rule: FlatRule);
  BEGIN
  Rule.FirstSet.Set(Code);
  END AddToFirstSet;

REDEFINE METHOD AddToLastSet (Rule: FlatRule);
  BEGIN
  Rule.LastSet.Set(Code);
  END AddToLastSet;

REDEFINE METHOD AddToFollowSet (Rule: FlatRule);
  BEGIN
  Rule.FollowSet.Set(Code);
  END AddToFollowSet;

REDEFINE METHOD AddToPrecedeSet (Rule: FlatRule);
  BEGIN
  Rule.PrecedeSet.Set(Code);
  END AddToPrecedeSet;

REDEFINE METHOD AddToMiddleSet (Rule: FlatRule;
                                LeftSkip,
                                RightSkip: BOOLEAN);
  BEGIN
  IF ¬ LeftSkip ∧ ¬ RightSkip THEN
    Rule.MiddleSet.Set(Code);
    END;
  END AddToMiddleSet;
```

We assume that `FlatTerminal`'s are `Smaller` than all other `FlatRuleComponent`-derived classes. When comparing two `FlatRuleComponents`, their codes are compared, and if equal, their attached target attributes are compared (See method *DuraFlat. FlatRuleComponent.Compare* on page 543).

547 ⟨`FlatTerminal` *class implementation* 545a⟩+≡

```
    REDEFINE METHOD Compare(Other: FlatRuleComponent): INTEGER;
      BEGIN
      WHAT Other OF
        IN FlatTerminal:
          IF Code > Other.Code THEN
            RESULT ← Greater;
           ELSIF Code < Other.Code THEN
            RESULT ← Smaller;
           ELSE
            RESULT ← BASE(Other);
            END;
          END;
       ELSE
        RESULT ← Smaller;
        END;
      END Compare;

  END FlatTerminal;
```

FlatTerminalSet are a mere generalization of FlatTerminal, in the sense that they denote a transition on a *set of* terminals instead of a single terminal. Keeping the two classes separate is merely a question of optimization, since handling these sets requires more processing power.

FlatTerminalSet

$FlatRuleComponent$ (See page 540)

$FlatTerminalSet$ is a generalization of $FlatTerminal$, in the sense that it denotes a terminal transition on a *set of codes* rather than on a single code only. The other properties listed hereunder are straightforward.

$Codes : \mathbb{P}\ TokenCode$ (See page 14)

$Closure = \{\delta Enclosing.Items(Position)\}$
$NullableLevel = NotNullable$
$Strings = \bigcup_{c \in Codes} \{\langle c \rangle\}$

548 ⟨FlatTerminalSet *class definition* 548⟩≡

```
CLASS FlatTerminalSet;
  INHERITS FlatRuleComponent;

  REDEFINE METHOD CREATE (Set: TerminalSet);
  METHOD GetTerminalSet: TerminalSet;
END FlatTerminalSet;
```

549a ⟨FlatTerminalSet *class implementation* 549a⟩≡
```
CLASS FlatTerminalSet;
  INHERITS FlatRuleComponent;

  VAR
    TheSet: TerminalSet;

  REDEFINE METHOD CREATE (Set: TerminalSet);
    BEGIN
    BASE;
    ASSERT Set ≠ Λ;
    ASSERT Set.UsedBits ≠ 0;
    TheSet ← Set;
    END CREATE;

  METHOD GetTerminalSet: TerminalSet;
    BEGIN
    RESULT ← TheSet;
    END GetTerminalSet;

  REDEFINE METHOD Dump (Output: OutputStream);
    BEGIN
    BASE (Output);
    TheSet.Dump (Output);
    END Dump;
```

Just as for FlatTerminal, a transition on a non-empty set of terminals is never nullable.

549b ⟨FlatTerminalSet *class implementation* 549a⟩+≡
```
REDEFINE METHOD NullableLevel: INTEGER;
  BEGIN
  RESULT ← FlatRule.CodeNotNullableLevel;
  END NullableLevel;
```

Quite predictable, I'd say.

549c ⟨FlatTerminalSet *class implementation* 549a⟩+≡
```
REDEFINE METHOD IsTerminalSet: BOOLEAN;
  BEGIN
  RESULT ← TRUE;
  END IsTerminalSet;
```

Instead of setting individual bits as for `FlatTerminals`, we `Or` sets here, but the basic idea remains basically the same.

550a ⟨`FlatTerminalSet` *class implementation* 549a⟩+≡

```
REDEFINE METHOD AddToFirstSet (Rule: FlatRule);
  BEGIN
  Rule.FirstSet.Or(TheSet);
  END AddToFirstSet;

REDEFINE METHOD AddToLastSet (Rule: FlatRule);
  BEGIN
  Rule.LastSet.Or(TheSet);
  END AddToLastSet;

REDEFINE METHOD AddToFollowSet (Rule: FlatRule);
  BEGIN
  Rule.FollowSet.Or(TheSet);
  END AddToFollowSet;

REDEFINE METHOD AddToPrecedeSet (Rule: FlatRule);
  BEGIN
  Rule.PrecedeSet.Or(TheSet);
  END AddToPrecedeSet;

REDEFINE METHOD AddToMiddleSet (Rule: FlatRule;
                                LeftSkip,
                                RightSkip: BOOLEAN);
  BEGIN
  IF ¬ LeftSkip ∧ ¬ RightSkip THEN
    Rule.MiddleSet.Or(TheSet);
    END;
  END AddToMiddleSet;
```

`Code` is redefined in such a way that two equivalent `FlatTerminalSets` (that is, denoting transitions on the same set of lexemes) will return the same code, even if we have a (small) probability of collision, that will not have any hard consequence beyond a slightly sub-optimal memory occupation.

550b ⟨`FlatTerminalSet` *class implementation* 549a⟩+≡

```
REDEFINE METHOD Code: INTEGER;
  BEGIN
  RESULT ← SYSTEM.ABS(FlatRuleComponent.TerminalSetOffset +
                 TheSet.HashValue);
  ⟨Make sure RESULT is kosher 551a⟩
  ASSERT RESULT ≥ FlatRuleComponent.TerminalSetOffset;
  END Code;
```

If, for any reason such as arithmetic overflow, `RESULT` as computed above is not valid for a `FlatTerminalSet`, its value is ajusted.

551a ⟨*Make sure* `RESULT` *is kosher* 551a⟩≡

```
WHILE RESULT < FlatRuleComponent.TerminalSetOffset DO
  RESULT ← SYSTEM.ABS(RESULT + FlatRuleComponent.TerminalSetOffset);
  END;
```

Comparing `FlatTerminalSets` is first based on the number of bits in the attached set of terminals in the transition. If the case where we are dealing with two otherwise non equal sets of the same size, we indicate that the two `FlatRuleComponents` are `AlmostEqual`, which means that they are not equal, but in absence of any finer-grained comparison criterion, they ought to be considered as equivalent by the sorting algorithm.

551b ⟨`FlatTerminalSet` *class implementation* 549a⟩+≡

```
REDEFINE METHOD Compare(Other: FlatRuleComponent): INTEGER;
  VAR
    MyBits, OtherBits: INTEGER;
  BEGIN
  RESULT ← Smaller;
  WHAT Other OF
    IN FlatTerminalSet:
      MyBits ← TheSet.UsedBits;
      OtherBits ← TAG.TheSet.UsedBits;
      IF MyBits < OtherBits THEN
        RESULT ← Smaller;
       ELSIF MyBits > OtherBits THEN
        RESULT ← Greater;
       ELSIF TheSet.IsEqual(TAG.TheSet) THEN
        RESULT ← BASE(Other);
       ELSE
         RESULT ← AlmostEqual;
         END;
      END;
    ⟨Comparing with other FlatRuleComponents 552a⟩
    END;
  END Compare;
```

When comparing `FlatTerminalSets` with other `FlatRuleComponents`, we use the following rule:

$$\forall\, t \in \mathit{FlatTerminal}, n \in \mathit{FlatNonTerminalSet}, \\ s \in \mathit{FlatTerminalSet} \bullet \\ t < s < n$$

552a ⟨*Comparing with other* `FlatRuleComponents` 552a⟩≡

```
IN FlatTerminal:
  RESULT ← Greater;
  END;
IN FlatNonTerminal:
  RESULT ← Smaller;
  END;
```

The following chunk implements the generation of a `yacc` equivalent grammar for a `FlatTerminalSet`. Unlike other forms of `FlatRuleComponent`, the equivalence toward a `yacc` -like form is not straightforward. It requires an intermediate grammar rule. For instance, if we deal with a grammar rule of the form

```
Ident [ Dot, Comma, SemiColon] Ident
```

in the flattened form, the `yacc` equivalent (as generated using the methods listed in this chunk) would be:

```
TC1 : Dot |
      Comma |
      SemiColon ;
N1 :  Ident TC1 N1 ;
```

These methods are called by `GenerateYacc` (See page 535).

552b ⟨`FlatTerminalSet` *class implementation* 549a⟩+≡

```
VAR
  TheNr: INTEGER;

METHOD GenerateYaccIdent (Output: OutputStream);
  VAR
    a: ARRAY OF CHAR;
    Counter: ONCE INTEGER;
  BEGIN
  IF TheNr = 0 THEN
    Counter ← Counter + 1;
    TheNr ← Counter;
    END;
  a ← ‘‘TC”+ IntConversions.IntToString (TheNr, 0);
  Output.WriteString (a);
  END GenerateYaccIdent;
```

GenerateYacc actually generates the ancillary non-terminal made of the disjunction of the various terminals in the terminal set.

553 ⟨FlatTerminalSet *class implementation* 549a⟩+≡

```
  METHOD GenerateYacc (Output: OutputStream);
    VAR
      p: ARRAY OF INTEGER;
    BEGIN
    GenerateYaccIdent (Output);
    Output.WriteLine (“:”);
    p ← TheSet.TokensRow;
    FOR i ∈ 0 .. p.SIZE - 1 DO
      IF i > 0 THEN
        Output.WriteLine (“|”);
        Output.WriteString (“  ”);
        END;
      Output.WriteString (DuraCfg.LexemeImage(p_i));
      END;
    Output.WriteLine (“;”);
    Output.WriteLn;
    END GenerateYacc;

END FlatTerminalSet;
```

FlatNonTerminal

FlatRuleComponent (See page 540)

FlatNonTerminal denotes a transition on a non-terminal, referred to by NT.

$NT : \rho FlatRule$

Whether a non-terminal transition is nullable or not depends on whether the *FlatRule* (See page 477) it refers to is nullable or not. In case of cyclic grammar rule, nullability is not relevant, and the grammar cannot produce a working parser anyway.

$\delta NT.Cycle = \text{true} \Rightarrow NullableLevel = Error$
$\delta NT.Cycle = \text{false} \Rightarrow NullableLevel = \delta NT.NullableLevel$

The possible derivation of a non-terminal transition can be just *itself*, that is, a string made of a single reference to the grammar rule it refers to ($\langle NT \rangle$) or one of the possible derivations of this rule ($\delta NT.Strings$).

$Strings = \{\langle NT \rangle\} \cup \delta NT.Strings$

The closure of a non-terminal transition contains the item *before* the transition itself, and the closures of the leftmost items of each of the right hand sides attached to the *FlatRule* (See page 477) it refers to.

$$Closure = \{\delta Enclosing.Items(Position)\} \cup \bigcup_{r \in \delta NT.RHS} (head(r.Items)).Closure$$

554a ⟨FlatNonTerminal *class definition* 554a⟩≡

```
CLASS FlatNonTerminal;
  INHERITS FlatRuleComponent;

  METHOD Rule: FlatRule;
  METHOD SetRule (Rule: FlatRule);

END FlatNonTerminal;
```

554b ⟨FlatNonTerminal *class implementation* 554b⟩≡

```
CLASS FlatNonTerminal;
  INHERITS FlatRuleComponent;

  VAR
    TheRule: FlatRule;

  ⟨Comparing two FlatNonTerminal instances 558a⟩
```

555a ⟨FlatNonTerminal *class implementation* 554b⟩+≡
```
    METHOD Rule: FlatRule;
      BEGIN
      RESULT ← TheRule;
      END Rule;

    METHOD SetRule (Rule: FlatRule);
      BEGIN
      TheRule ← Rule;
      END SetRule;
```

555b ⟨FlatNonTerminal *class implementation* 554b⟩+≡
```
  REDEFINE METHOD Dump (Output: OutputStream);
    BEGIN
    BASE (Output);
    Output.WriteChar (“(”);
    IF TheRule.Class ≠ Λ THEN
      Output.WriteString (TheRule.Class.DotImage + “:”);
      END;
    Output.WriteInt (TheRule.Nr, 0);
    Output.WriteChar (“)”);
    END Dump;
```

The nullability level and the reducibility attached to a `FlatNonTerminal` are deduced from the grammar rule it refers to.

555c ⟨FlatNonTerminal *class implementation* 554b⟩+≡
```
  REDEFINE METHOD NullableLevel: INTEGER;
    BEGIN
    RESULT ← TheRule.NullableLevel;
    END NullableLevel;

  REDEFINE METHOD Reducible: BOOLEAN;
    BEGIN
    RESULT ← TheRule.Reducible;
    END Reducible;

  REDEFINE METHOD AlwaysNull: BOOLEAN;
    BEGIN
    RESULT ← TheRule.AlwaysNull;
    END AlwaysNull;
```

The closure of a `FlatNonTerminal` is also deduced from its attached rule, but it is worth mentioning that actually calling `Closure` for `FlatRule` (See page 491) does *not* compute the closure, which must have been computed beforehand using a global process (See method *DuraFlSpec.FlatSpecification.ComputeClosures* on page 567).

556a ⟨`FlatNonTerminal` *class implementation* 554b⟩+≡
```
REDEFINE METHOD Closure: FlatItemSet;
  BEGIN
  RESULT ← TheRule.Closure;
  END Closure;
```

`Code` is redefined in such a way that it does lie between `RuleOffset` and `TerminalSetOffset` (See page 541).

556b ⟨`FlatNonTerminal` *class implementation* 554b⟩+≡
```
REDEFINE METHOD Code: INTEGER;
  BEGIN
  RESULT ← Rule.Nr + FlatRuleComponent.RuleOffset;
  END Code;
```

`IsNonTerminal` (See page 544) is redefined in a rather straightforward way . . .

556c ⟨`FlatNonTerminal` *class implementation* 554b⟩+≡
```
REDEFINE METHOD IsNonTerminal: BOOLEAN;
  BEGIN
  RESULT ← TRUE;
  END IsNonTerminal;
```

Handling the sets for a non-terminal transition is merely a question of `Or`ing with the `FirstSet` (See page 504) or `LastSet` (See page 504) of the attached rule, depending on whether the set we are computing is at the left or at the right of the component.

557 ⟨`FlatNonTerminal` *class implementation* 554b⟩+≡

```
    REDEFINE METHOD AddToFirstSet (Rule: FlatRule);
      BEGIN
      Rule.FirstSet.Or(THIS.Rule.FirstSet);
      END AddToFirstSet;

    REDEFINE METHOD AddToLastSet (Rule: FlatRule);
      BEGIN
      Rule.LastSet.Or(THIS.Rule.LastSet);
      END AddToLastSet;

    REDEFINE METHOD AddToFollowSet (Rule: FlatRule);
      BEGIN
      Rule.FollowSet.Or(THIS.Rule.FirstSet);
      END AddToFollowSet;

    REDEFINE METHOD AddToPrecedeSet (Rule: FlatRule);
      BEGIN
      Rule.PrecedeSet.Or(THIS.Rule.PrecedeSet);
      END AddToPrecedeSet;

    REDEFINE METHOD AddToMiddleSet (Rule: FlatRule;
                                     LeftSkip,
                                     RightSkip: BOOLEAN);
      BEGIN
      IF ¬ LeftSkip THEN
        Rule.MiddleSet.Or (THIS.Rule.FirstSet);
        END;
      IF ¬ RightSkip THEN
        Rule.MiddleSet.Or (THIS.Rule.LastSet);
        END;
      Rule.MiddleSet.Or (THIS.Rule.MiddleSet);
      END AddToMiddleSet;

  END FlatNonTerminal;
```

558a ⟨*Comparing two* `FlatNonTerminal` *instances* 558a⟩≡
```
REDEFINE METHOD Compare(Other: FlatRuleComponent): INTEGER;
  BEGIN
  WHAT Other OF
    IN FlatNonTerminal:
      IF Code > Other.Code THEN
        RESULT ← Greater;
      ELSIF Code < Other.Code THEN
        RESULT ← Smaller;
      ELSE
        RESULT ← BASE(Other);
        END;
      END;
   ELSE
    RESULT ← Greater;
    END;
  END Compare;
```

7.13.12 Additional classes `TerminalSet` and `NonTerminalSet`

`TerminalSet` is just a bit map, where the only application-specific added service is the `Dump` method that lists the non-terminals in textual form on an output stream.

558b ⟨`TerminalSet` *class definition* 558b⟩≡
```
CLASS TerminalSet;
  INHERITS BitMap;

  REDEFINE METHOD CREATE;

  METHOD Dump (Output: OutputStream);
  METHOD TokensRow: ARRAY OF INTEGER;
END TerminalSet;
```

559a ⟨`TerminalSet` *class implementation* 559a⟩≡

```
  CLASS TerminalSet;
    INHERITS BitMap;

    REDEFINE METHOD CREATE;
      BEGIN
      BASE (DuraCfg.LastToken);
      END CREATE;

    ⟨Dumping a TerminalSet 559b⟩

    METHOD TokensRow: ARRAY OF INTEGER;
      VAR
        Index: INTEGER;
      BEGIN
      RESULT ← ALL i ∈ 0 .. Size − 1 • THIS_i;
      END TokensRow;

  END TerminalSet;
```

559b ⟨*Dumping a* `TerminalSet` 559b⟩≡

```
  METHOD Dump (Output: OutputStream);
    VAR
      Count: INTEGER;
    BEGIN
    Output.WriteString ("{");
    ⟨Dump all the lexemes in THIS 559c⟩
    Output.WriteLine ("}");
    END Dump;
```

559c ⟨*Dump all the lexemes in* `THIS` 559c⟩≡

```
  FOR i ∈ 0 .. Size−1 | THIS_i DO
    Count ← Count + 1;
    IF Count = 8 THEN
      Output.WriteLn;
      Output.WriteString ("  ");
      Count ← 0;
      END;
    Output.WriteString (DuraCfg.LexemeImage(i) + " ");
    END;
```

Just like `TerminalSet` presented above, `NonTerminalSet` is also just a bit map. It provides no higher level services, it is the client code's responsibility to turn rules into their corresponding number to be able to use these bitmaps.

560a ⟨`NonTerminalSet` *class definition* 560a⟩≡

```
CLASS NonTerminalSet;
  INHERITS BitMap;

  REDEFINE METHOD CREATE(FlatSpec: FlatSpecification);

END NonTerminalSet;
```

560b ⟨`NonTerminalSet` *class implementation* 560b⟩≡

```
CLASS NonTerminalSet;
  INHERITS BitMap;

  REDEFINE METHOD CREATE(FlatSpec: FlatSpecification);
    BEGIN
    BASE (FlatSpec.Size);
    END CREATE;

END NonTerminalSet;
```

7.13.13 Mundane stuff

560c ⟨*Definition import list* 560c⟩≡

```
IMPORT ArrayList;
IMPORT BitMap;
FROM DuraClDefinition IMPORT DuraClassDefinition;
FROM DuraFlSpec IMPORT FlatSpecification;
FROM DuraIntermediateFormat IMPORT InterRule, InterAttribute;
FROM DuraItem IMPORT FlatItem, FlatItemSet;
FROM DuraOutput IMPORT DuraOutputStream;
FROM DuraPDA IMPORT PushDownAutomaton, PDAState;
FROM DuraReductions IMPORT DuraReduction;
FROM DuraTransitions IMPORT NonTerminalTransitionList;
FROM Streams IMPORT OutputStream;
```

561 ⟨*Implementation import list* 561⟩≡

```
FROM DuraBitMap IMPORT DuraBitMap;
FROM DuraCfg IMPORT DuraCfg;
FROM DuraReductions IMPORT FetchNonTerminalReduction;
FROM DuraTransitions IMPORT TerminalTransition, NonTerminalTransition;

FROM Conversions IMPORT IntConversions;
FROM Streams IMPORT StdOut;
IMPORT SYSTEM;

FROM List IMPORT List;
IMPORT Set;
IMPORT Stack;
```

7.14 DuraFlSpec

The `DuraFlSpec` module implements a grammar specification in its flattened form (See module *DuraFlat* on page 474).

⟨*Definition* ⟩≡
```
DEFINITION MODULE DuraFlSpec;

⟨Definition import list 618a⟩
```

`FlatSpecification` is the only class of this module. It is a list that contains the `FlatRules` of the specification (See page 480). Purists might argue against this inheritance relationship since it implements notational comfort when compared with aggregation, but definitely no subtyping. However, I like it this way, and it is consistent with `FlatRule` (See page 480) which is a list of `FlatRHS` (See page 513), which in turn is a list of `FlatComponent` (See page 540), etc.

Besides, the fact that `FlatSpecification` inherits from `ArrayList`, hence, from `RandomCollection` allows us to use quantifiers on its enclosed `FlatRules`, and this module makes extensive use of this facility.

⟨*Definition* ⟩+≡
```
  CLASS FlatSpecification;
    INHERITS ArrayList(FlatRule);

  METHOD Dump (Output: OutputStream);
  METHOD Renumber;
  METHOD ComputeClosures;

  ⟨Removing unreachable rules, definition 565a⟩
  ⟨Normalizing, definition 568⟩
  ⟨Computing rule properties, definition 598b⟩
  ⟨Computing production sets, definition 601a⟩
  ⟨Computing FIRST and FOLLOW sets, definition 602c⟩
  ⟨Handling precedence and associativity, definition 573c⟩

  ⟨Compute PDA transitions, definition 574b⟩
```

continued...

```
⟨Definition ⟩+≡
    ⟨Generate reduction methods, definition 582a⟩
    ⟨Generate goto methods, definition 582c⟩
    ⟨Produce an interpreted goto array, definition 582e⟩

    ⟨GenerateRHSToRule method definition 585e⟩
    ⟨GenerateFetchAttributes method definition 586b⟩
    ⟨GenerateIsVoidMethod method definition 592a⟩
    ⟨Recovery table management, definition 593b⟩
    ⟨Various table management, definition 596e⟩

    ⟨GenerateGenericGotoMethod method definition 590d⟩
    ⟨GenerateGenericReduceMethod method definition 591b⟩

    ⟨Root rule handling, definition 604b⟩

    ⟨Closure and rule inclusion handling, definition 606a⟩

    ⟨FinalItem method definition 607b⟩
    ⟨Merging equivalent rules, definition 608⟩
    ⟨LastRHSNr method definition 614d⟩
    ⟨Error handling, definition 614b⟩

    END FlatSpecification;

  END DuraFlSpec;
```

⟨*Implementation* ⟩≡
```
IMPLEMENTATION MODULE DuraFlSpec;

⟨Implementation import list 618b⟩

  CLASS FlatSpecification;
    INHERITS ArrayList(FlatRule);


  ⟨Numbering methods 613b⟩

  ⟨ComputeClosures method implementation 567c⟩
  ⟨Removing unreachable rules, implementation 565b⟩
  ⟨Normalizing, implementation 569a⟩
  ⟨Handling precedence and associativity, implementation 574a⟩
  ⟨Compute PDA transitions, implementation 575a⟩
  ⟨Computing rule properties, implementation 599a⟩
  ⟨Computing production sets, implementation 601b⟩
  ⟨Computing FIRST and FOLLOW sets, implementation 603a⟩

  ⟨Generate reduction methods, implementation 582b⟩
  ⟨Generate goto methods, implementation 582d⟩
  ⟨Produce an interpreted goto array, implementation 583⟩
  ⟨Recovery table management, implementation 593c⟩
  ⟨Various table management, implementation 597⟩

  ⟨GenerateRHSToRule method implementation 586a⟩
  ⟨GenerateFetchAttributes method implementation 587a⟩
  ⟨GenerateIsVoidMethod method implementation 592b⟩
  ⟨GenerateGenericGotoMethod method implementation 591a⟩
  ⟨GenerateGenericReduceMethod method implementation 591c⟩

  ⟨Root rule handling, implementation 604c⟩
  ⟨Closure and rule inclusion handling, implementation 606b⟩
  ⟨FinalItem method implementation 607c⟩
  ⟨Merging equivalent rules, implementation 609a⟩
  ⟨Ensuring transitivty of the comparison function 617⟩

  ⟨LastRHSNr method implementation 615⟩
  ⟨Error handling, implementation 614c⟩
```

⟨*Implementation* ⟩+≡
```
    METHOD Dump (Output: OutputStream);
      BEGIN
      FOR El ∈ THIS DO
        IF El = Root THEN
          Output.WriteString (“ROOT: ”);
          END;
        El.Dump (Output);
        END;
      END Dump;

    END FlatSpecification;

  END DuraFlSpec;
```

7.14.1 Removing unreachable grammar rules

When working on grammar rules, specially in DURA's very generic and simplistic way, it often happens that grammar rules are simply not reachable anymore because the production that did refer to them has been deleted or altered by some form of normalization or optimization process.

Furthermore, since DURA deals with full YAFL projects, if the project contains more than one parser (typically, with different Root grammar rules), grammar rules for one parser will not be taken into consideration when generating the other.

The RemoveUnreachableRules method starts with the Root rule of our grammar, marks reachable rules, and delete the others. It basically works as a mark & sweep garbage collector.

565a ⟨*Removing unreachable rules, definition* 565a⟩≡
```
  METHOD RemoveUnreachableRules;
```

565b ⟨*Removing unreachable rules, implementation* 565b⟩≡
```
  METHOD RemoveUnreachableRules;
    VAR
      ToVisit: Stack(FlatRule);
      Visited: Set(FlatRule);
      TheRule: FlatRule;
    BEGIN
    ⟨Clear counts 566c⟩
```

`ToVisit` keeps track of the rules that must still be visited. `Visited` keeps track of the rules that have been visited or that will eventually be visited because they are registered in `ToVisit`.

Both the stack `ToVisit` and the set `Visited` are initialized with the root grammar rule, from which the set of all reachable rules will be computed.

566a ⟨*Removing unreachable rules, implementation* 565b⟩+≡
```
ToVisit.CREATE;
Visited.CREATE;
Visited.Add(Root);
ToVisit.Push(Root);
```

as long as there is something to do, go on...

566b ⟨*Removing unreachable rules, implementation* 565b⟩+≡
```
WHILE ToVisit.Depth > 0 DO
  TheRule ← ToVisit.Pop;
  ⟨Visit all the rules directly reachable from TheRule 566d⟩
  END;
⟨Remove the rules that have not been visited 567b⟩
END RemoveUnreachableRules;
```

We take advantage of this visit through all grammar rules to recompute usage counts. Hence, we first clear them (See method *DuraFlat.FlatRule.ClearUsageCount* on page 487), and the usage count attached to the proper grammar rule will be incremented as we visit a non-terminal reference to this grammar rule.

566c ⟨*Clear counts* 566c⟩≡
```
FOR Rule ∈ THIS DO
  Rule.ClearUsageCount;
  END;
```

We increment the usage count for every rule we encounter (See method *DuraFlat.FlatRule.IncUsageCount* on page 487).

566d ⟨*Visit all the rules directly reachable from* `TheRule` 566d⟩≡
```
FOR RHS ∈ TheRule DO
  FOR FlatEl ∈ RHS DO
    WHAT FlatEl OF
      IN FlatNonTerminal:
        ASSERT TAG.Rule ≠ Λ;
        TAG.Rule.IncUsageCount;
        ⟨Consider whether TAG.Rule should be visited 567a⟩
        END;
     ELSE
      -- Don't abort...
      END;
    END;
  END;
```

`Visited` is a set that keeps track of which rules have been visited already.

567a ⟨*Consider whether* `TAG.Rule` *should be visited* 567a⟩≡

```
IF Visited.Cardinality (TAG.Rule) = 0 THEN
  Visited.Add(TAG.Rule);
  ToVisit.Push(TAG.Rule);
  END;
```

We could have decided whether the rule was used or not using the usage counts as well. Well, we rather use the `Visited` set. Not that it makes any kind of significant difference though. Since we wish to be able to delete elements while going through the list, we scan it backwards (`BY -1`).

567b ⟨*Remove the rules that have not been visited* 567b⟩≡

```
FOR i ∈ Size-1 .. 0 BY -1 | Visited.Cardinality(THIS_i = 0 DO
  Delete (i);
  END;
Renumber;
```

`ComputeClosures` compute closures (you might have guessed it that far). It uses a very simple and efficient scheme: it recomputes the closures repeatedly until they are stable. `ComputeClosure` as defined for `FlatRule` (See page 491) returns a number of bits that grows every time an element is added to a closure.

567c ⟨`ComputeClosures` *method implementation* 567c⟩≡

```
METHOD ComputeClosures;
  VAR
    OldTotal, Total: INTEGER;
  BEGIN
  ⟨First, make sure all items have been allocated 567d⟩
  OldTotal ← -1;
  WHILE Total ≠ OldTotal DO
    OldTotal ← Total;
    Total ← 0;
    FOR Rule ∈ THIS DO
      Total ← Total + Rule.ComputeClosure;
      END;
    END;
  END ComputeClosures;
```

This dirty part, that simply consults the first item in every `FlatRHS` of every `FlatRule` in the specification (See method *DuraFlat.FlatRHS.GetItem* on page 516), just to make sure they are allocated. This is needed to allocated the closure's bitmaps which size is determined by the number of such items that are in existence in the system.

567d ⟨*First, make sure all items have been allocated* 567d⟩≡

```
FOR Rule ∈ THIS DO
  FOR RHS ∈ Rule DO
    Λ ← RHS.GetItem(0);
    END;
  END;
```

7.14.2 Normalizing the grammar

Normalization is a complex topic. It deals with the ability to produce the most compact and efficient representation of a grammar without changing any of its semantics. For instance, consider the following example:

$$
\begin{array}{lcl}
a & ::= & b \mid \\
  &   & c \\
c & ::= & d \mid \\
  &   & e \\
e & ::= & f \mid \\
  &   & g
\end{array}
$$

We can all agree that this grammar would be strictly equivalent to:

$$
\begin{array}{lcl}
a & ::= & b \mid \\
  &   & d \mid \\
  &   & f \mid \\
  &   & g
\end{array}
$$

Well, it's not that simple. For instance, one must keep in mind that we don't deal with grammars only, but also with semantic actions. when replacing (one might say "expanding") $a ::= c$ by $a ::= d$ and $a ::= e$, we also replace the reduction that might have been attached to $a ::= c$ by another one. In many cases, this is perfectly acceptable. When $a ::= c$ implements the fact that c is represented by a class that is derived from a, the default behaviour of replacement of a rule by another is perfectly acceptable. On the other hand, if it represented a common grammar rule, where c might be fetched onto an attribute, this fetching operation must be retained.

Quite obviously, it is the attached reduction (See class *DuraReductions.DuraReduction* on page 822) that determines whether the rule can be fooled around with safely.

One might argue that such complex patterns as those described here do not occur in practice. Well, in fact, they do. At least, in DURA's context, they obviously do. The difference comes from the fact that the intermediate (flattened) representation which is being used here is generated automatically based on a more expressive representation. Hence, there are redundancies that can appear here that would not be likely if the grammar was to be written by a human coder.

568 ⟨*Normalizing, definition* 568⟩≡

```
METHOD Normalize;
```

Normalization is an iterative process, that stops as soon as no rule can be removed from the grammar without altering its original sense. In other words, this method will perform replacements in grammar rules, making some of them obsolete (unused), until no more grammar rule can be made obsolete.

One might question this algorithm's stability. Given N_i the number of grammar rules that remain within the grammar at iteration number i, would it be possible to have a j and a $k > j$ such as $N_j = N_{j+1}$ and $N_k < N_j$?

Well, I don't know, and I don't really care. We try to optimize, hoping to improve things, even if we neglect a small area of possible improvements. What we are doing here does not affect the system's semantics in any way, it is merely a way of simplifying the grammar.

`Normalize` first considers whether the user has explicitely asked *not* to normalize the parser (See module *DuraCfg* on page 370).

569a ⟨*Normalizing, implementation* 569a⟩≡
```
METHOD Normalize;
  VAR
    OldSize: INTEGER;
    OtherRHS: FlatRHS;
    OtherRule: FlatRule;
    WinningReduction: DuraReduction;
    ReductionConflict: BOOLEAN;
    OldTarget: InterAttribute;
  BEGIN
  RemoveUnreachableRules;
  IF DuraCfg.NormalizeParser THEN
    WHILE (OldSize ≠ Size) DO
      OldSize ← Size;
      ⟨One normalization pass 569b⟩
      END;
    END;
  END Normalize;
```

A normalization pass basically normalizes all the rules in the specification, then removes rules that might have been made unreachable by the normalization, by calling `RemoveUnreachableRules` (See page 565).

569b ⟨*One normalization pass* 569b⟩≡
```
FOR Rule ∈ THIS DO
  ⟨Normalize Rule 569c⟩
  END;
RemoveUnreachableRules;
```

and normalizing a rule is the same thing as normalizing each of its right-hand sides.

569c ⟨*Normalize* `Rule` 569c⟩≡
```
FOR RHS ∈ Rule DO
  ⟨Normalize RHS by pasting uniquely used external rules 570a⟩
  IF (RHS.Size = 1) ∧ (RHS.First.Target = Λ) THEN
    ⟨Normalize RHS by replacement if applicable 571⟩
    END;
  END;
```

This chunk deals with the ability to replace a reference to a rule by a body (right hand side), which would produce a more efficient parser since we bypass a reduction.

This operation is only possible when several conditions are met. First, the rule we refer to must state that it can be inlined (See method *DuraFlat.FlatRule.Inlinable* on page 484). Then, we must also make sure that it is ok for the rule we are considering to “receive” the other rule’s body, to be sure that there are no incompatibilities.

570a ⟨*Normalize* `RHS` *by pasting uniquely used external rules* 570a⟩≡

```
FOR k ∈ RHS.Size - 1 .. 0 BY -1 DO
  WHAT RHS_k OF
    IN FlatNonTerminal:
      OtherRHS ← TAG.Rule.Inlinable;
      IF OtherRHS ≠ Λ THEN
        ⟨Try to paste OtherRHS into RHS at position k 570b⟩
        END;
      END;
     ELSE
    END;
  END;
```

These incompatibilities can be of three forms:

- There is a target involved
- There is a mismatch in recovery policies
- There is a mismatch in snip level

570b ⟨*Try to paste* `OtherRHS` *into* `RHS` *at position* `k` 570b⟩≡

```
IF (TAG.Target = Λ) ∨ (OtherRHS.Size = 1) THEN
  IF (OtherRHS.CutLevel = 0) ∨
     (OtherRHS.CutLevel = RHS.CutLevel) THEN
    IF (OtherRHS.RecoveryLevel = 0) ∨
       (OtherRHS.RecoveryLevel = RHS.RecoveryLevel) THEN
      OldTarget ← TAG.Target;
      ⟨Paste OtherRHS at position k within RHS 570c⟩
      ASSERT (OldTarget ≠ Λ) ⇒
       (RHS_k.Target = OldTarget);
      END;
    END;
  END;
```

Pasting is implemented by `ReplaceComponent` (See page 524).

570c ⟨*Paste* `OtherRHS` *at position* `k` *within* `RHS` 570c⟩≡

```
RHS.ReplaceComponent (k, OtherRHS);
```

We’ll refer to normalization by replacement if several conditions are met:

- The RHS we are considering is made of a singe non-terminal reference

$$\begin{array}{lcl} a & ::= & b \mid \\ & & c\ d\ e \end{array}$$

The first RHS $a ::= b$ matches this criterion if b denotes a non-terminal.

- That non-terminal must support a single RHS itself. Otherwise, no pasting can occur, since several alternatives might have to be taken into account. For instance, in:

$$\begin{array}{lcl} a & ::= & b \\ b & ::= & c \mid \\ & & d \end{array}$$

b's body cannot be pasted in $a ::= b$ because b is made of more than a single right hand side. Pasting it would require duplicating s's right hand sides.

- There are no conflicts between the various reductions, that is, one of the two reductions must be implicit or void, in which case the other one takes over.

- That single RHS's length is 1, or the usage count of the other rule is 1.

This last criterion is meant to avoid a dramatic growth of the grammar by excessive inlining. Consider the following example:

$$\begin{array}{lcl} A & ::= & abcdefghiujklmnopqrstuvxyz \\ B & ::= & A \\ C & ::= & A \\ D & ::= & A \\ & \ldots & \\ Z & ::= & A \end{array}$$

If the usage count criterion was not used, inlining as described here would transform this grammar into:

$$\begin{array}{lcl} A & ::= & abcdefghiujklmnopqrstuvxyz \\ B & ::= & abcdefghiujklmnopqrstuvxyz \\ C & ::= & abcdefghiujklmnopqrstuvxyz \\ D & ::= & abcdefghiujklmnopqrstuvxyz \\ & \ldots & \\ Z & ::= & abcdefghiujklmnopqrstuvxyz \end{array}$$

which is clearly much larger than the original grammar, if not in terms of the number of resulting states, in terms of the number of items.

571 ⟨*Normalize* RHS *by replacement if applicable* 571⟩≡

```
WHAT RHS.First OF
  IN FlatNonTerminal:
    OtherRule ← TAG.Rule;
```

`OtherRule`'s usage count must be greater than zero, since there is at least a reference to it. It is precisely this reference that we try to optimize by pasting `OtherRule`'s body.

572a ⟨*Normalize* `RHS` *by replacement if applicable* 571⟩+≡

```
ASSERT OtherRule.UsageCount > 0;
```

572b ⟨*Normalize* `RHS` *by replacement if applicable* 571⟩+≡

```
      IF (OtherRule.Size = 1) ∧ (OtherRule.UsageCount = 1) THEN
        OtherRHS ← OtherRule.First;
        ⟨Determine which reduction from RHS or OtherRHS should be kept 572c⟩
        ⟨Make sure that there is no conflict in cut and recovery levels 573a⟩
        IF ¬ ReductionConflict THEN
          ⟨All the conditions are met, replace now !!! 573b⟩
          END;
        END;
    END;
  ELSE
   END;
```

The following chunk takes two reductions `RHS.Reduction` and `OtherRHS.Reduction` and chooses the reduction that should be retained for the resulting RHS, and sets `WinningReduction`. If there is no way one can remove one of the reductions to the advantage of the other, `ReductionConflict` is set to `TRUE` and the merge is cancelled.

572c ⟨*Determine which reduction from* `RHS` *or* `OtherRHS` *should be kept* 572c⟩≡

```
ReductionConflict ← FALSE;
WinningReduction ← Λ;
IF (RHS.Reduction = Λ) ∨ RHS.Reduction.Implicit THEN
  WinningReduction ← OtherRHS.Reduction;
 ELSIF (OtherRHS.Reduction = Λ) ∨
              OtherRHS.Reduction.Implicit THEN
  WinningReduction ← RHS.Reduction;
 ELSE
  ReductionConflict ← TRUE;
  END;
```

Even if the two reductions have no intrinsic incompatibility that prevents the merge from happening, the two rules cannot be merged if they differ in terms of they cut or recovery levels.

573a ⟨*Make sure that there is no conflict in cut and recovery levels* 573a⟩≡
```
IF ¬ ReductionConflict THEN
  IF (OtherRHS.CutLevel ≠ 0) ∧
     (OtherRHS.CutLevel ≠ RHS.CutLevel) ∧
     (RHS.CutLevel ≠ 0) THEN
   ReductionConflict ← TRUE;
  ELSIF (OtherRHS.RecoveryLevel ≠ 0) ∧
        (OtherRHS.RecoveryLevel ≠ RHS.RecoveryLevel) ∧
        (RHS.RecoveryLevel ≠ 0) THEN
   ReductionConflict ← TRUE;
   END;
  END;
```

The replacement in itself is amazingly simple: we simply empty `RHS`, and copy the content of `OtherRHS` into it. Then, we attach the winning reduction, the cut and the recovery level to `RHS`.

573b ⟨*All the conditions are met, replace now !!!* 573b⟩≡
```
ASSERT RHS.Size = 1;
ASSERT RHS.First.Target = Λ;
RHS.Purge;
FOR El ∈ OtherRHS DO
  RHS.Append (El.CLONE);
  END;
RHS.SetReduction (WinningReduction);
RHS.SetCutLevel (OtherRHS.CutLevel);
RHS.SetRecoveryLevel (OtherRHS.RecoveryLevel);
```

7.14.3 Handling precedence and associativity

573c ⟨*Handling precedence and associativity, definition* 573c⟩≡
```
METHOD HandleAssociativity ;
```

Handling associativity means:

- Normalizing (See method *DuraFlSpec.FlatSpecification.Normalize* on page 568)
- Having each rule checking its own associativity and precedence level (See method *DuraFlat.FlatRule.HandleAssociativity* on page 492)
- Closing the grammar (See method *DuraFlSpec.FlatSpecification.Close* on page 606) to ensure that the grammar rules that were newly created by associativity and precedence handling at the `FlatRule` level (See above) are included in the flat specification
- Normalizing again, to possibly perform optimizations that were made possible by the various handlings listed above.

574a ⟨*Handling precedence and associativity, implementation* 574a⟩≡

```
METHOD HandleAssociativity;
  BEGIN
  Normalize;
  FOR Rule ∈ THIS DO
    Rule.HandleAssociativity;
    END;
  Close;
  Normalize;
  END HandleAssociativity;
```

7.14.4 Turning a flat specification into a PDA

574b ⟨*Compute PDA transitions, definition* 574b⟩≡

```
METHOD ComputeAutomaton(Output: OutputStream):
                                  PushDownAutomaton ;
```

The algorithm presented here is amazingly similar to the one described for lexical analyzers (See method *DfaAutomaton.DfaAutomaton.CREATE* on page 129) We start with an initial state, and then build new states by following all possible transitions, adding newly created states or connecting to existing states until all transitions have been handled. All the states are denoted by the `PDAState` class (See page 636), and are grouped together in a `PushDownAutomaton` (See page 651).

We maintain a stack `ToVisit` that keeps track of all states we must still visit, and which is initialized with the start state of our new push-down automaton.

575a ⟨*Compute PDA transitions, implementation* 575a⟩≡

```
METHOD ComputeAutomaton (Output: OutputStream):
                                        PushDownAutomaton ;
  CONST
    ThresholdStep = 256;
  VAR
    Res: PushDownAutomaton;
    First, Last, NextThreshold: INTEGER;
    NonTermTranList, TempList: NonTerminalTransitionList;
    StartState, CurState: PDAState;
    ToVisit: Stack (PDAState);

    ⟨VisitState local method 576b⟩

  BEGIN
  ComputeClosures;
  ⟨Compute StartState 575b⟩
  ⟨Initialize ToVisit and Res with StartState 575c⟩
  NextThreshold ← ThresholdStep;
  WHILE ToVisit.Depth > 0 DO
    ⟨Dump something if required 576a⟩
    CurState ← ToVisit.Pop;
    VisitState(CurState);
    END;
  RESULT ← Res;
  ⟨Attach all non-terminal transitions to their transiting symbol 581b⟩
  RESULT.Renumber(FinalItem);
  StdOut.WriteInt (RESULT.Size, 0);
  StdOut.WriteLine (“ states generated”);
  END ComputeAutomaton;
```

`StartState` is made of the closure of the items that start the right-hand sides of the `Root` grammar rule.

575b ⟨*Compute* `StartState` 575b⟩≡

```
StartState.CREATE(Root.Closure);
```

`ToVisit` must receive `StartState` as initial element (for the loop that polls `ToVisit` to execute at least once), and so must `Res` because `StartState` is a state of our push down automaton.

575c ⟨*Initialize* `ToVisit` *and* `Res` *with* `StartState` 575c⟩≡

```
ToVisit.CREATE;
ToVisit.Push (StartState);
Res.CREATE;
Res.Add (StartState);
```

576a ⟨*Dump something if required* 576a⟩≡

```
IF (DuraCfg.VerboseLevel > 0) ∧ (Res.Size ≥ NextThreshold) THEN
  StdOut.WriteInt (NextThreshold, 0);
  StdOut.WriteLine (“ states generated”);
  NextThreshold ← NextThreshold + ThresholdStep;
  END;
```

`VisitState` is invoked for each state of our push down automaton, computes all the outgoing transitions, and connects these transitions to existing states, or to newly created states if required. `VisitState` also attaches the reductions that can take place when in this state.

576b ⟨`VisitState` *local method* 576b⟩≡

```
METHOD VisitState (CurState: PDAState);
  VAR
    TargetSet: FlatItemSet;
    TerminalSetTransitionList,
    SortedList: ArrayList(FlatItem);
    TargetState: PDAState;
    CurrentCode,
    Index, First, Last: INTEGER;
    TermTran: TerminalTransition;
    NonTermTran: NonTerminalTransition;
    TheSet, VisitedTerminals: TerminalSet;
  BEGIN
  VisitedTerminals.CREATE;
  ⟨Obtain SortedList 577a⟩
  ⟨Consider FarRight items 577b⟩
  ⟨Consider terminal set transitions 578a⟩
  ⟨Handle transitions that go from CurState 578c⟩
  END VisitState;
```

First, obtain `SortedList` that denotes the list of all the items in `CurState`, by calling `SortedByTransition` (See page 646). `SortedList` is then sorted using an ad hoc comparison function (See method *DuraItem.FlatItemList.ShouldSwap* on page 629) which is based on a double rule:

- The `FarRight` (See page 625) items are put first in any circumstances. These items are *not* followed by a `FlatComponent`, that is, applying `NextComponent` (See page 626) to them yields void. Consequently, such items indicate the possibility of performing a reduce operation. They identify the rightmost positions in grammar rules.

- Among items that are not `FarRight` items, the comparison function uses the result of the `Code` (See method *DuraFlat.FlatRuleComponent.Code* on page 541) method as sorting criterion.

We shall use this sorted property extensively in the following chunks.

577a ⟨*Obtain* `SortedList` 577a⟩≡

```
SortedList ← CurState.SortedByTransition;
```

A reduction transition is characterized by the fact that its items are on the far right position within their enclosing rule. Since the sorting method ensures that `SortedList` will have such items first, we simply scan this list as long as the items we find are indeed far right items (See method *DuraItem.FlatItem.FarRight* on page 625).

For every such far right item in `CurState`, we tell `CurState` that a reduction can occur on the right hand side of which this item is the far right extremity (See method *DuraPDA.PDAState.AppendReduction* on page 647).

577b ⟨*Consider* `FarRight` *items* 577b⟩≡

```
WHILE (SortedList.Size > 0) ∧ SortedList.First.FarRight DO
  CurState.AppendReduction (SortedList.First.RHS);
  SortedList.Delete(0);
  END;
```

All far right items in `CurrentState` were placed in front of `SortedList`. They have been taken care of, and removed. The remaining items in `SortedList` are not far right items any longer.

Let's now build `TerminalSetTransitionList`, that includes all the items that are placed before terminal set transitions, that is, transitions for which a set of terminal symbol has been defined as opposed to terminal transitions for which a single terminal symbol is accepted.

By the way the `Code` method is implemented for such transitions, and by the fact that what remains of the original `SortedList` is sorted by increasing `Code` value, we know for sure that all items that refer to terminal set transitions are at the end of `SortedList`.

578a ⟨*Consider terminal set transitions* 578a⟩≡

```
TerminalSetTransitionList.CREATE;
Index ← SortedList.Size - 1;
WHILE (Index ≥ 0) ∧
      SortedList_Index.NextComponent.IsTerminalSet DO
  TerminalSetTransitionList.Append (SortedList_Index;
  SortedList.Delete (Index);
  Index ← Index - 1;
  END;
```

Ensure there is no terminal set transition left...

578b ⟨*Consider terminal set transitions* 578a⟩+≡

```
ASSERT ∀Item ∈ SortedList •
  ¬ Item.NextComponent.IsTerminalSet;
```

Now, what remains in `SortedList` are the non-far right items which are followed by a `FlatTerminal` or a `FlatNonTerminal`. Since `SortedList` is sorted, all the transitions related to a symbol (terminal or non-terminal) are grouped together.

The following chunk goes through `SortedList`, identifies sub-lists that relate to a single symbol (by using `First` and `Last` as pointers to define the sub-list, while `CurrentCode` denotes the symbol).

578c ⟨*Handle transitions that go from* `CurState` 578c⟩≡

```
Index ← 0;
WHILE Index < SortedList.Size DO
  ⟨Compute CurrentCode, First and Last 579a⟩
  ⟨Build TargetSet for all items from First to Last 579b⟩
  IF CurrentCode < FlatRuleComponent.RuleOffset THEN
    ⟨Add terminal sets transitions matching CurrentCode to TargetSet 581a⟩
    VisitedTerminals.Set(CurrentCode);
    END;
  ⟨Assign TargetState, create if required 579c⟩
  ⟨Build a CurrentCode-transition from CurState to TargetState 580a⟩
  Index ← Last + 1;
  END;
```

This chunk computes `First` and `Last` in such a way that all the items in `SortedList` for indexes that go from `First` to `Last` refer to the same symbol, that is, the same terminal or the same non-terminal. This is of course made possible by the fact that `SortedList` is sorted accordingly. Well, what can one expect from a variable called `SortedList` anyway...

CurrentCode is just an integer variable that holds the code for all the items that go from First to Last. It is only later in the processing (see the chunks below) that we will bother about what this code actually represents.

579a ⟨*Compute* CurrentCode, First *and* Last 579a⟩≡

```
CurrentCode ← SortedList_Index.NextComponent.Code;
First ← Index;
Last ← Index;
WHILE (Last+1 < SortedList.Size) ∧
      (SortedList_Last+1.NextComponent.Code = CurrentCode) DO
  Last ← Last + 1;
  END;
```

TargetSet is the closure of all the items that follow items found in SortedList, that is, items that refer to positions in the grammar rule after that the symbol (terminal or non-terminal) has been parsed.

579b ⟨*Build* TargetSet *for all items from* First *to* Last 579b⟩≡

```
TargetSet.CREATE;
FOR i ∈ First .. Last DO
  TargetSet.Or (SortedList_i;
  END;
```

Using TargetSet, we now check whether we already have a PDA state in Res that matches this TargetSet. If no such state can be found, a new state is created, added to the push-down automaton, and pushed onto ToVisit to ensure that at some stage, the outgoing transition from this newly created state will be examined.

579c ⟨*Assign* TargetState, *create if required* 579c⟩≡

```
TargetState ← Res.Exists (TargetSet);
IF TargetState = Λ THEN
  TargetState.CREATE (TargetSet);
  Res.Add (TargetState);
  ToVisit.Push (TargetState);
  END;
```

Depending on whether `CurrentCode` refers to a terminal or a non-terminal transition, we create a `TerminalTransition` (See page 661) or a `NonTerminal-Transition` (See page 662), which is then added to `CurState` (See method*DuraPDA.PDAState.AppendTerminalTransition* on page 647 and*DuraPDA.PDAState.AppendNonTerminalTransition* on page 647).

580a ⟨*Build a* `CurrentCode`*-transition from* `CurState` *to* `TargetState` 580a⟩≡

```
IF CurrentCode < FlatRuleComponent.RuleOffset THEN
  TermTran.CREATE (CurState, TargetState, CurrentCode);
  CurState.AppendTerminalTransition (TermTran);
 ELSE
  ASSERT CurrentCode < FlatRuleComponent.TerminalSetOffset;
  NonTermTran.CREATE (CurState, TargetState,
        THIS.Get (CurrentCode MOD FlatRuleComponent.RuleOffset));
  CurState.AppendNonTerminalTransition (NonTermTran);
  END;
```

When encountering a transition on a terminal t, we also check `TerminalTransitionList` for transitions on terminal sets that contained t, and t is added to `VisitedTerminals` to indicate that all the possible transitions starting at this state on t have been taken into account. Now, we must also consider all the terminal set transitions that have not been visited yet, that is, considering all the terminals that are not in `VisitedTerminals`.

580b ⟨*Handle transitions that go from* `CurState` 578c⟩+≡

```
FOR t ∈ TerminalSetTransitionList DO
  WHAT t.NextComponent OF
    IN FlatTerminalSet:
      TheSet ← TAG.GetTerminalSet;
      ⟨Check all tokens in TheSet 580c⟩
      END;
    END;
  END;
```

580c ⟨*Check all tokens in* `TheSet` 580c⟩≡

```
FOR j ∈ 0 .. TheSet.Size - 1 |
   TheSet_j ∧ ¬ VisitedTerminals_j DO
  CurrentCode ← j;
  TargetSet.CREATE;
  ⟨Add terminal sets transitions matching CurrentCode to TargetSet 581a⟩
  ⟨Assign TargetState, create if required 579c⟩
  ⟨Build a CurrentCode-transition from CurState to TargetState 580a⟩
  VisitedTerminals.Set (j);
  END;
```

581a ⟨*Add terminal sets transitions matching* `CurrentCode` *to* `TargetSet` 581a⟩≡

```
FOR Item ∈ TerminalSetTransitionList DO
  WHAT Item.NextComponent OF
    IN FlatTerminalSet:
      IF TAG.GetTerminalSet_CurrentCode THEN
        TargetSet.Or(Item.Next.Closure);
        END;
      END;
    END;
  END;
```

The following chunk is not part of the computing of the PDA per se, but it is closely related in terms of data structure. The purpose here is to go through all the non-terminal transitions, and attach them to the corresponding non-terminal (See method *DuraFlat. FlatRule.SetTransitionsOnMe* on page 498). For instance, the non-terminal `Statement` will be attached to all transitions where one goes from one state onto another by accepting a `Statement`.

The information produced here will be used to build the `goto` transition table (as described in ) that control the state one goes to after a reduce operation.

581b ⟨*Attach all non-terminal transitions to their transiting symbol* 581b⟩≡

```
ASSERT NonTermTranList = Λ;
NonTermTranList ← RESULT.NonTerminalTransitions;
NonTermTranList.SetSortMethod (NonTermTranList.ByTransitionSortMethod);
NonTermTranList.Sort;
First ← 0;
WHILE First < NonTermTranList.Size DO
  Last ← First;
  ⟨Compute the Last index corresponding to the same rule 581c⟩
  TempList.CREATE;
  FOR i ∈ First .. Last DO
    TempList.Append (NonTermTranList_i;
    END;
  TempList.First.Rule.SetTransitionsOnMe (TempList);
  First ← Last + 1;
  END;
```

Since `NonTermTranList` is sorted by transition (See above), all the transitions on the same rule are grouped together...

581c ⟨*Compute the* `Last` *index corresponding to the same rule* 581c⟩≡

```
WHILE (Last+1 < NonTermTranList.Size) ∧
  (NonTermTranList_First.Rule = NonTermTranList_Last+1 DO
  Last ← Last + 1;
  END;
```

7.14.5 Code generation

GenerateReduceMethods generates one reduce method for every right-hand side in the grammar, each of these methods being called when the corresponding right hand side is being reduced.

582a ⟨*Generate reduction methods, definition* 582a⟩≡

```
METHOD GenerateReduceMethods (Output: DuraOutputStream);
```

Basically, GenerateReduceMethods works by calling GenerateReduceMethodBody (See page 526) for every right hand side for every rule in the grammar.

582b ⟨*Generate reduction methods, implementation* 582b⟩≡

```
METHOD GenerateReduceMethods (Output: DuraOutputStream);
  BEGIN
  FOR Rule ∈ THIS DO
    FOR Rhs ∈ Rule DO
      Rhs.GenerateReduceMethodBody (Output);
      END;
    END;
  END GenerateReduceMethods;
```

582c ⟨*Generate goto methods, definition* 582c⟩≡

```
METHOD GenerateGotoMethods (Output: DuraOutputStream);
```

GenerateReduceMethods generates one goto method for every grammar rule in the system. These methods implement the *goto* function as described in  in case of a compiled parser, StoreGotoTable (See the facing page) being used in case of an interpreted parser.

The methods are actually generated by GenerateGotoMethodBody, as provided by FlatRule (See page 502).

582d ⟨*Generate goto methods, implementation* 582d⟩≡

```
METHOD GenerateGotoMethods (Output: DuraOutputStream);
  BEGIN
  FOR Rule ∈ THIS DO
    Rule.GenerateGotoMethodBody (Output);
    END;
  END GenerateGotoMethods;
```

582e ⟨*Produce an interpreted goto array, definition* 582e⟩≡

```
METHOD StoreGotoTable (Dict: IntDictionary;
                       AutomatonSize: INTEGER): INTEGER;
```

StoreGotoTable is the interpreted counterpart of GenerateGotoMethods described above. It includes no intelligence besides calling GotoArray (See page 503), and merging those of the so-computed *goto* arrays that can be merged. It is used in the parser skeleton (See method *Parser skeleton.Parser skeleton.GenericGoto* on page 874).

583 ⟨*Produce an interpreted goto array, implementation* 583⟩≡

```
METHOD StoreGotoTable (Dict: IntDictionary;
                       AutomatonSize: INTEGER): INTEGER;

  ⟨Merge utility methods 585b⟩

  VAR
    Temp: ARRAY OF ARRAY OF INTEGER;
    Arr: ARRAY OF INTEGER;
    MergeCount,
    k: INTEGER;
  BEGIN
  Temp.CREATE (Size);
  Temp_0 ← THIS_0;
  FOR i ∈ 1 .. Size - 1 DO
    Arr ← THIS_i;
    ⟨Consider merging Arr with a previously computed array 584a⟩
    END;
  ⟨Dump merge stats 585d⟩
  RESULT ← Dict.StoreArrayArray(Temp);
  END StoreGotoTable;
```

The very idea of merging, as explained (briefly) in , comes as follows. The *goto* function takes two parameters: a state s and a non-terminal n, and returns the state one must go to if reducing the non-terminal n and if the state the system was in before starting to analyse n was s.

Let's first define $goto_n$ as the one-parameter function one gets when restricting *goto* to a non-terminal n. A possible implementation of $goto_n$ as an array of integers indexed by state number is computed by the GotoArray method (See page 503).

The problem comes from the fact that such a naive implementation is very memory consuming: it allocates an array indexed by state number for every non-terminal in the grammar. Large parsers have commonly over 1500 states, and over 1000 flattened rules, which amounts to a total of 1500000 integers to implement the *goto* function, which is quite excessive.

Fortunately, we can often merge two or more of such $goto_n$ arrays in a single array, thereby using far less memory. Merging happens by understanding how the $goto_n$ functions are used. When reducing a n, we check the state s we were in before starting to analysing the n, and use the $goto_n$ function to decide which state we must go to. On the other hand, this only occurs if *it is acceptable* to have a n when in state s. In a state s', that cannot be followed by a n, we don't even start recognizing a n, so we don't ever reduce a n either, and consequently, we never even evaluate $goto_n(s')$. There is no need to put error codes in $goto_n$ because the erroneous cases never occur. We only evaluate $goto_n$ in states where $goto_n$ yields something meaningful.

So, when two non-terminals n_1 and n_2 have attached $goto$ functions that yield the same results on the (possibly empty) intersection of their domains, we can implemented the two $goto$ functions in a single array.

The following chunk is implemented naively. It checks every new array that implements a $goto_n$ function against all the arrays that have been created so far for possible merges. It merges `Arr` with the first compatible array it can find in `Temp`, or adds it to `Temp` otherwise.

This scheme delivers excellent results (over 90% of the $goto_n$ functions are merged this way), but one could still question whether this scheme is optimal. In other words, would another algorithm increase the number of merges ? Can an algorithm be optimal in terms of the number of arrays that are left over after that merges are performed ?

The answer is simple: yes, there is an optimal algorithm for this problem, but the only implementations we can think of today are exponential by nature, since this is a slight variation on a notoriously np-complete problem. Formally, we can state our problem as follows. We have a set S and an incompatibility relationship $R \subset S^2$ where $(s_1, s_2) \in R \Rightarrow s_1 \neq s_2$. The problem is to build a partition $P_i, i = 1..n$ such that $\forall\, s_1, s_2 \in P_i \bullet (s_1, s_2) \notin R$, and to minimize n in the process. If n is a number of colors, and if R describes the common border relationship, this problem is the map colouring problem, which is notoriously np-complete [5].

584a ⟨*Consider merging* `Arr` *with a previously computed array* 584a⟩≡

```
k ← FIRST Idx ∈ 0 .. i-1 • Mergeable (Arr, Temp_Idx);
IF (k ≥ 0) ∧ (k < Temp.SIZE) THEN
  DEBUG
    MergeCount ← MergeCount + 1;
    END;
  ⟨Do merge 584b⟩
 ELSE
  ⟨Don't merge 585a⟩
  END;
```

584b ⟨*Do merge* 584b⟩≡

```
Merge (Temp_k, Arr);
```

The following assignment does not allocate a new array: it is a reference assignment, so that elements with indexes i and k refer to a single shared array.

584c ⟨*Do merge* 584b⟩+≡

```
Temp_i ← Temp_k;
```

`Arr` cannot be merged with another array in `Temp`.

585a ⟨*Don't merge* 585a⟩≡
```
Temp_i ← Arr;
```

Deciding whether two arrays (understand: two *goto* functions attached to different grammar rules) can be merged is implemented by the `Mergeable` method, while the actual merge is implemented by the `Merge` method.

585b ⟨*Merge utility methods* 585b⟩≡
```
METHOD Mergeable (One, Two: ARRAY OF INTEGER): BOOLEAN;
  BEGIN
  ASSERT One ≠ Λ;
  ASSERT Two ≠ Λ;
  ASSERT One.SIZE = Two.SIZE;
  RESULT ← ∀ i ∈ 0 .. One.SIZE - 1 •
    ((One_i = Two_i) ∨ (One_i = 0) ∨ (Two_i = 0));
  END Mergeable;
```

585c ⟨*Merge utility methods* 585b⟩+≡
```
METHOD Merge (Base, New: ARRAY OF INTEGER);
  BEGIN
  ASSERT Base ≠ Λ;
  ASSERT New ≠ Λ;
  ASSERT Base.SIZE = New.SIZE;
  FOR i ∈ 0 .. Base.SIZE - 1 | New_i ≠ 0 DO
    Base_i ← New_i;
    END;
  END Merge;
```

585d ⟨*Dump merge stats* 585d⟩≡
```
DEBUG
  StdOut.WriteInt (MergeCount, 0);
  StdOut.WriteString ("/");
  StdOut.WriteInt (Size, 0);
  StdOut.WriteString (" (");
  StdOut.WriteInt (AutomatonSize, 0);
  StdOut.WriteLine (") merged in goto table");
  END;
```

585e ⟨`GenerateRHSToRule` *method definition* 585e⟩≡
```
METHOD GenerateRHSToRule (Output: DuraOutputStream);
```

`GenerateRHSToRule` produce the body of a `CASE` statement that returns the rule number given a right-hand side number, as in:

```
CASE RhsNr OF
  0,1,2:
    RESULT := 0;
    END;
  3,4:
    RESULT := 1;
    END;
  5:
    RESULT := 2;
    END;
  6,7,8:
    RESULT := 3;
    END;
  ...
```

586a ⟨`GenerateRHSToRule` *method implementation* 586a⟩≡

```
  METHOD GenerateRHSToRule (Output: DuraOutputStream);
    BEGIN
    FOR Rule ∈ THIS DO
      FOR j ∈ 0 .. Rule.Size - 1 DO
        IF j > 0 THEN
          Output.WriteString (", ");
          END;
        Output.WriteInt (Rule_j.Nr, 0);
        END;
      Output.WriteLine (":");
      Output.IncIndent (2);
      Output.WriteString ("RESULT := ");
      Output.WriteInt (Rule.Nr, 0);
      Output.WriteLine (";");
      Output.End;
      END;
    END GenerateRHSToRule;
```

586b ⟨`GenerateFetchAttributes` *method definition* 586b⟩≡

```
  METHOD GenerateFetchAttributes (Output: DuraOutputStream);
```

`GenerateFetchAttributes` produces the body of the `FetchAttributes` method as generated in the skeleton (See page 884) as a number of `CASE` tags, that refer to right hand side numbers.

587a ⟨`GenerateFetchAttributes` *method implementation* 587a⟩≡

```
METHOD GenerateFetchAttributes (Output: DuraOutputStream);

  VAR
    ToVisitIndex: INTEGER;
    TheTarget: InterAttribute;
    TagGenerated: BOOLEAN;

    ⟨Emit CASE tag utility method 587b⟩

  BEGIN
  FOR Rule ∈ THIS DO
    FOR RHS ∈ Rule DO
      TagGenerated ← FALSE;
      ⟨Fetch attributes one by one 588⟩
      ⟨Consider recursive plunge for each of RHS components 590a⟩
      ⟨Consider TheRHS's reduction 590c⟩
      IF TagGenerated THEN
        Output.End;
        END;
      END;
    END;
  END GenerateFetchAttributes;
```

In order to avoid generating hundreds `CASE` tags for values that don't require any action, that is, for right hand sides where no attribute must be fetched, the `CASE` tag is generated on demand.

587b ⟨*Emit* `CASE` *tag utility method* 587b⟩≡

```
METHOD EmitCaseTag (RHS: FlatRHS);
  BEGIN
  IF ¬ TagGenerated THEN
    TagGenerated ← TRUE;
    Output.WriteInt (RHS.Nr, 0);
    Output.WriteLine (“:”);
    Output.IncIndent (2);
    END;
  END EmitCaseTag;
```

Fetching the attributes is implemented by calling `GenerateFetch` (See page 740) for all the components in `RHS` that have a target.

588 ⟨*Fetch attributes one by one* 588⟩≡

```
FOR k ∈ 0 .. RHS.Size - 1 DO
  TheTarget ← RHS_k.Target;
  IF TheTarget ≠ Λ THEN
    EmitCaseTag (RHS);
    Output.WriteString (“    -- ”);
    Output.WriteInt (k, 0);
    Output.WriteLine (“  ” + TheTarget.FullName);
    TheTarget.GenerateFetch (Output, Index ← k);
    END;
  END;
```

All sub-rules that refer to other classes must have been fetched themselves. On the other hand, sub-rules that do not refer to classes are synthetic rules that have been generated when flattening the grammar, and assignments that occur within such sub-rules must be executed.

A small optimization is required here: if implemented too naively, this recursive plunge will require a recursive call for each element in a list aggregated using the “*” or “+” grammatical operators. Hence, in order to be far more scalable, we use a brute force derecursification.

This derecursification does not apply to DURA itself, but rather, to DURA-generated parsers. While the technique is pretty straightforward to code, it is slightly more difficult to generate code that uses this technique.

We use a degenerated form of tail-recursion optimisation. When the visiting method applied to an object calls itself for a number of sub-objects, the last of these sub-objects is stored into a variable, and a loop reexecutes the method's body on the last sub-object without requiring an extra recursive call. The other sub-objects are visited normally, using a recursive plunge. By construction, such recursive plunge should be of limited depth.

So, instead of having something such as:

```
METHOD FetchAttributes(Root: ParserStackElement;
                       Nt: BaseNonTerminal);
VAR
  TheArr: ARRAY OF ParserStackElement;
BEGIN
...
CASE RhsNr OF
  ...
  72:
    FetchAttributes (TheArr[3]);
    FetchAttributes (TheArr[1]);
    FetchAttributes (TheArr[0]);
    END;
  ...
```

we produce something of the form:

```
METHOD FetchAttributes(Root: ParserStackElement;
                       Nt: BaseNonTerminal);
VAR
  TheArr: ARRAY OF ParserStackElement;
BEGIN
ToVisit := Root;
WHILE ToVisit <> VOID DO
  ...
  CASE RhsNr OF
    ...
    72:
      FetchAttributes (TheArr[3]);
      FetchAttributes (TheArr[1]);
      ToVisit := TheArr[0];
      END;
    ...
  END;
```

Since `DURA` basically generates a LR parser, the iteration operators "`*`" and "`+`" produce left-recursive rules that use less LR stack space than the corresponding right-recursive rules (See method *DuraIntermediateFormat.InterRepetitionExpression.BuildFlatReference* on page 732). Hence, we assume that the leftmost sub-rule will be more likely to be recursive. We visit the various sub-rules in a right-to-left fashion, and store the last (hence, the leftmost) sub-rule in a variable named `ToVisit` instead of performing the recursive plunge directly.

The skeleton will then visit `ToVisit` by iteration rather than recursion (See method *Parser skeleton.Parser skeleton.FetchAttributes* on page 884).

590a ⟨*Consider recursive plunge for each of* RHS *components* 590a⟩≡
```
ToVisitIndex ← -1;
FOR k ∈ RHS.Size - 1 .. 0 BY -1 DO
  WHAT RHS_k OF
    IN FlatNonTerminal:
      IF TAG.Rule.Class = Λ THEN
        IF ToVisitIndex ≥ 0 THEN
          EmitCaseTag(RHS);
          Output.WriteString ("FetchAttributes (TheArr[");
          Output.WriteInt (ToVisitIndex, 0);
          Output.WriteLine ("], Nt);");
          END;
        ToVisitIndex ← k;
        END;
      END;
   ELSE
    -- Don't abort, could be a TerminalComponent
    END;
  END;
```

If ToVisitIndex is non negative, a recusive plunge for the last attribute to be fetched must be replaced by a loop iteration. The following chunk generates code that extracts the element from the array to fetch TheArr at index ToVisitIndex, and stores it into ToVisit.

590b ⟨*Consider recursive plunge for each of* RHS *components* 590a⟩+≡
```
IF ToVisitIndex ≥ 0 THEN
  EmitCaseTag(RHS);
  Output.WriteString ("ToVisit := TheArr[");
  Output.WriteInt (ToVisitIndex, 0);
  Output.WriteLine ("];");
  END;
```

After fetching attributes, we check whether there is a reduction object attached to RHS, and if there is, we call its code generation method (See method *DuraReductions. DuraReduction.GenerateFetchCode* on page 823).

590c ⟨*Consider* TheRHS*'s reduction* 590c⟩≡
```
IF RHS.Reduction ≠ Λ THEN
  EmitCaseTag (RHS);
  RHS.Reduction.GenerateFetchCode (Output, RHS);
  END;
```

590d ⟨GenerateGenericGotoMethod *method definition* 590d⟩≡
```
METHOD GenerateGenericGotoMethod(Output: DuraOutputStream);
```

`GenerateGenericGotoMethod` generates the body of the `GenericGoto` method (See page 874). This method takes a rule number and a state number as parameters, and calls the goto method (See method *DuraFlat.FlatRule.GotoMethodName* on page 500) attached to this rule. The only optimization here lies in the usage of `UniqueTargetOnMe` (See page 499) to return a constant target state if the goto function attached to this rule returns the same value on its entire domain.

This will typically be the case for rules that occur only once in a grammar. Consequenly, there is a single state the parser can be in after reducing this rule.

591a ⟨`GenerateGenericGotoMethod` *method implementation* 591a⟩≡

```
METHOD GenerateGenericGotoMethod(Output: DuraOutputStream);
  BEGIN
  FOR Rule ∈ THIS DO
    Output.WriteInt (Rule.Nr, 0);
    Output.WriteLine (‘‘:”);
    Output.IncIndent(2);
    Output.WriteString (‘‘RESULT := ”);
    IF Rule.UniqueTargetOnMe ≠ Λ THEN
      Output.WriteInt (Rule.UniqueTargetOnMe.Nr, 0);
     ELSE
      Output.WriteString (Rule.GotoMethodName + ‘‘(PreviousState)”);
      END;
    Output.WriteLine (‘‘;”);
    Output.End;
    END;
  END GenerateGenericGotoMethod;
```

591b ⟨`GenerateGenericReduceMethod` *method definition* 591b⟩≡

```
METHOD GenerateGenericReduceMethod(Output: DuraOutputStream);
```

`GenerateGenericReduceMethod` generates the body of the `GenericReduce` method (See page 875). `GenericReduce` does not actually perform the reduction itself, it merely calls the additional code that must be executed beside the reduction, depending on the kind of reduction which is attached to RHS (See method *DuraFlat.FlatRHS.RequiresAdHocReduction* on page 526).

591c ⟨`GenerateGenericReduceMethod` *method implementation* 591c⟩≡

```
METHOD GenerateGenericReduceMethod(Output: DuraOutputStream);
  BEGIN
  FOR Rule ∈ THIS DO
    FOR RHS ∈ Rule | RHS.RequiresAdHocReduction DO
      Output.WriteInt (RHS.Nr, 0);
      Output.WriteLine (‘‘:”);
      Output.IncIndent(2);
      RHS.GenerateReduceMethodCall (Output);
      Output.End;
      END;
    END;
  END GenerateGenericReduceMethod;
```

592a ⟨GenerateIsVoidMethod *method definition* 592a⟩≡
```
METHOD GenerateIsVoidMethod (Output: DuraOutputStream);
```

`GenerateIsVoidMethod` generates the body of the `IsVoid` method (See page 878) as a number of CASE tags on the right hand side numbers. `GenerateIsVoidMethod` produces code for grammar rules which are known to be nullable.

592b ⟨GenerateIsVoidMethod *method implementation* 592b⟩≡
```
METHOD GenerateIsVoidMethod (Output: DuraOutputStream);
  BEGIN
  FOR Rule ∈ THIS DO
    FOR RHS ∈ Rule | RHS.Nullable DO
      ⟨Generate the code to check whether RHS is void 592c⟩
      END;
    END;
  END GenerateIsVoidMethod;
```

First, we check whether the grammar rule is empty, in which case `IsVoid` must always return `TRUE`.

592c ⟨*Generate the code to check whether* RHS *is void* 592c⟩≡
```
Output.WriteInt (RHS.Nr, 0);
Output.WriteLine (":");
Output.IncIndent (2);
IF RHS.Size = 0 THEN
  Output.WriteLine ("RESULT := TRUE;");
 ELSE
```

Otherwise, we just compute the conjunction of the nullability of the various components of the reduction we are testing. These is no need for any derecursification here, since the nesting depth of nullable grammar rules is limited. This is a direct consequence of the fact that nullable grammar rules cannot be repeated. For instance, if n is a nullable non-terminal, that is, such that $n \overset{*}{\Rightarrow} \epsilon$, $n+$ is implemented by a synthetic grammar rule n', defined by

$$\begin{array}{lll} n' & ::= & n' \quad n \mid \\ & & n \end{array}$$

and since n is nullable, we have $n' \stackrel{*}{\Rightarrow} n'$, n' is in a grammatical cycle, which is detected and forbidden by DURA.

593a ⟨*Generate the code to check whether* RHS *is void* 592c⟩+≡

```
    Output.WriteLine (“Arr := TAG.Array;”);
    Output.WriteString (“RESULT := ”);
    Output.IncIndent (6);
    FOR k ∈ 0 .. RHS.Size - 1 DO
      IF k > 0 THEN
        Output.WriteLine (“ AND”);
        END;
      Output.WriteString (“IsVoid(Arr[”);
      Output.WriteInt (k, 0);
      Output.WriteString (“])”);
      END;
    Output.DecIndent (6);
    Output.WriteLine (“;”);
    END;
  Output.End;
```

7.14.6 Recovery table management

DURA's strategy when it comes to error recovery are described extensively when discussing the implementation details in the skeleton (See method *Parser skeleton.Parser skeleton.Recover* on page 900). The chunks listed here merely implement the generation of the tables used when performing error recovery.

593b ⟨*Recovery table management, definition* 593b⟩≡

```
  METHOD StoreRecoveryInfo (Dict: IntDictionary;
                            Automaton: PushDownAutomaton);
```

593c ⟨*Recovery table management, implementation* 593c⟩≡

```
  VAR
    TheSyncTable,
    TheRuleNrTable,
    TheRecoverTable: ARRAY OF ARRAY OF INTEGER;
```

`StoreRecoveryInfo` takes a `FlatSpecification` in `THIS`, an automaton (See page 651) and an `IntDictionary` (See page 764), and builds error recovery information and stores it into `Dict`.

594a ⟨*Recovery table management, implementation* 593c⟩+≡

```
METHOD StoreRecoveryInfo (Dict: IntDictionary;
                          Automaton: PushDownAutomaton);
  VAR
    Terms: TerminalSet;
    Nr, Code: INTEGER;
    RList, RuleList: ArrayList(FlatRHS);
    StateList: ArrayList(PDAState);
    SyncArr, StateArr: ARRAY OF PDAState;
    TheSyncStates: ARRAY OF BitMap;
    TheBitMap: BitMap;
  BEGIN
  IF TheRecoverTable = Λ THEN
    ⟨First, gather all rules for which a RECOVER set is defined in RList 594b⟩
    ⟨Then, consider all tokens that might be found in a RECOVER set 594c⟩
    ⟨Finally, normalize the tables by storing them into Dict 596a⟩
    END;
  END StoreRecoveryInfo;
```

594b ⟨*First, gather all rules for which a RECOVER set is defined in* `RList` 594b⟩≡

```
RList.CREATE;
Terms.CREATE;
FOR Rule ∈ THIS DO
  FOR RHS ∈ Rule | RHS.RecoverSet ≠ Λ DO
    RList.Append (RHS);
    Terms.Or (RHS.RecoverSet);
    END;
  END;
```

`Terms` holds the set of all terminals that are in the `RecoverSet` of a right hand side. Our entire strategy depends on the current token, so `TheRecoverTable` and `TheRuleNr` tables are indexed by token.

594c ⟨*Then, consider all tokens that might be found in a RECOVER set* 594c⟩≡

```
TheRecoverTable.CREATE (Terms.Size);
TheRuleNrTable.CREATE (Terms.Size);
TheSyncStates.CREATE (Automaton.Size);
FOR i ∈ 0 .. Terms.Size - 1 | Terms_i DO
  ⟨Consider token i 595a⟩
  END;
```

595a ⟨*Consider token* i 595a⟩≡

```
StateList.CREATE;
RuleList.CREATE;
FOR RHS ∈ RList | RHS.RecoverSet_i ∧ (RHS.Size > 0) DO
  ⟨Consider RHS, with i ∈ Recover(RHS) 595b⟩
  END;
ASSERT StateList.Size = RuleList.Size;
ASSERT StateList.Size > 0;
TheRecoverTable_i.CREATE (StateList.Size);
TheRuleNrTable_i.CREATE (StateList.Size);
FOR j ∈ 0 .. StateList.Size - 1 DO
  TheRecoverTable_i,j ← StateList_j.Nr;
  TheRuleNrTable_i,j ← RuleList_j.GetFlatRule.Nr;
  END;
```

We can recover on `RHS` if there is a state on the LR stack that includes its left-most item (`RHS.GetItem(0)`), and if the next state on the lr stack includes an item that follows `RHS.GetItem(0)` by one transition (shift or reduce). `RHS.GetItem(0).Closure.Reachable(1)` returns the set of such items.

`StatesByItem` (See page 656) returns all the states in an automaton that include a given item, while `StatesByItemSet` (See page 656) returns all the states in an automaton that have a non-empty intersection include a given set of items.

595b ⟨*Consider* RHS, *with* $i \in$ *Recover*(RHS) 595b⟩≡

```
StateArr ← Automaton.StatesByItem (RHS.GetItem(0));
SyncArr ← Automaton.StatesByItemSet (RHS.GetItem(0).Closure.
                        Reachable(1));
IF (StateArr ≠ Λ) ∧ (SyncArr ≠ Λ) THEN
  FOR State ∈ StateArr DO
    RuleList.Append (RHS);
    StateList.Append (State);
    Nr ← State.Nr;
    IF TheSyncStates_Nr = Λ THEN
      TheSyncStates_Nr.CREATE(Automaton.Size + 1);
      END;
    FOR PDAState ∈ SyncArr DO
      TheSyncStates_Nr.Set(PDAState.Nr);
      END;
    END;
  END;
```

Normalizing an array of array of integers means ensuring that every array of integer is only stored once, even if it is used numerous times in the two dimension array (See method *DuraIntDictionary.IntDictionary.StoreArrayArray* on page 783). In other words, if two entries in the two dimentional array hold the same value, that is, the same array of integer, it will be allocated only once.

`SaveName` (See page 785) is then used to attach a name to the stored array of array of integer, so that it can be retrieved conveniently afterwards.

Finally, after *saving* `TheRecoverTable` in the `IntDictionary` (and thereby, normalizing it as well), we extract the normalized value, and store it back into `TheRecoverTable` so that from now on, we use the normalized – understand: space-optimized – version of this recovery table.

596a ⟨*Finally, normalize the tables by storing them into* `Dict` 596a⟩≡
```
Code ← Dict.StoreArrayArray(TheRecoverTable);
Dict.SaveName (''Recover'', Code);
TheRecoverTable ← Dict.GetArrayArray (Code);
```

We proceed similarly with `TheRuleNrTable` . . .

596b ⟨*Finally, normalize the tables by storing them into* `Dict` 596a⟩+≡
```
Code ← Dict.StoreArrayArray(TheRuleNrTable);
Dict.SaveName (''RecoverRuleNr'', Code);
TheRuleNrTable ← Dict.GetArrayArray (Code);
```

while we must first compute `TheSyncTable` before storing it into the `IntDictionary`. `SyncStates` is an array of `BitMaps`, while the only kind of data structure we can store into an `IntDictionary` are arrays of integers, or array of arrays of integers, etc.

596c ⟨*Finally, normalize the tables by storing them into* `Dict` 596a⟩+≡
```
TheSyncTable.CREATE (TheSyncStates.SIZE);
FOR i ∈ 0 .. TheSyncStates.SIZE - 1 DO
  TheBitMap ← TheSyncStates_i;
  IF TheBitMap ≠ Λ THEN
    TheSyncTable_i ←
      ⟨Convert TheBitMap into an array of integers 596d⟩;
    END;
  END;
Code ← Dict.StoreArrayArray(TheSyncTable);
Dict.SaveName (''RecoverSync'', Code);
TheSyncTable ← Dict.GetArrayArray (Code);
```

YAFL We use a YAFL quantifier to build the array of all x's such that the x^{th} bit is turned on in `TheBitMap`.

596d ⟨*Convert* `TheBitMap` *into an array of integers* 596d⟩≡
```
ALL x ∈ 0 .. TheBitMap.Size - 1 • TheBitMap_x
```

596e ⟨*Various table management, definition* 596e⟩≡
```
METHOD LengthTable: ARRAY OF INTEGER;
METHOD NullableTable: ARRAY OF INTEGER;
METHOD RHSToRuleNrTable: ARRAY OF INTEGER;
```

- `LengthTable` returns an array that gives the length of each right hand side in the system.
- `NullableTable` returns an array that indicates whether a right hand side is nullable or not (which cannot be deduced from the rule's length only). Rather than defining `NullableTable` as an array of booleans, it is defined as an array of integers (0 = false, 1 = false) so that it can be stored in an `IntDictionary`.
- `RHSToRuleNrTable` returns an array that maps a right hand side number to the corresponding rule number.

These arrays are computed lazily, that is, on the first query, and stored in ad hoc attributes for later retrieval.

597 ⟨*Various table management, implementation* 597⟩≡

```
VAR
  TheLengthTable,
  TheNullableTable,
  TheRHSToRuleNrTable: ARRAY OF INTEGER;

METHOD ComputeTables;
  VAR
    Index: INTEGER;
    Rule: FlatRule;
  BEGIN
  IF TheLengthTable = Λ THEN
    TheLengthTable.CREATE (LastRHSNr+1);
    TheNullableTable ← TheLengthTable.CLONE;
    TheRHSToRuleNrTable ← TheLengthTable.CLONE;
    FOR i ∈ 0 .. Size-1 DO
      Rule ← THIS_i;
      FOR RHS ∈ Rule DO
        Index ← RHS.Nr;
        TheLengthTable_Index ← RHS.Size;
        IF ¬ RHS.Nullable THEN
          TheNullableTable_Index ← 1;
          END;
        TheRHSToRuleNrTable_Index ← i;
        END;
      END;
    END;
  END ComputeTables;
```

Based on ComputeTables (see above), the following methods are trivial.

598a ⟨*Various table management, implementation* 597⟩+≡
```
METHOD LengthTable: ARRAY OF INTEGER;
  BEGIN
  ComputeTables;
  RESULT ← TheLengthTable;
  END LengthTable;

METHOD NullableTable: ARRAY OF INTEGER;
  BEGIN
  ComputeTables;
  RESULT ← TheNullableTable;
  END NullableTable;

METHOD RHSToRuleNrTable: ARRAY OF INTEGER;
  BEGIN
  ComputeTables;
  RESULT ← TheRHSToRuleNrTable;
  END RHSToRuleNrTable;
```

The properties that can be computed for the grammar rules are:

- Nullability. Can the rule be reduced by an empty string ?
- Reducability. Can the rule be reduced by a finite string ? Nullability clearly implies reducibility

At first sight, it might be tempting to think of nullability as a local issue, that is, it should be possible to decide whether a grammar rule is nullable or not without considering the entire grammar[4]. In cases of excessive recursion, one could have preferred to take just a conservative approach, and considered that the rule was not nullable, assuming that one would not take any risk in doing so.

This approximate approach is wrong. Nullability is an essential issue. $FOLLOW$ and $FIRST$ sets are based on correct nullability. Considering a non-nullable rule as nullable might mean that we perform a reduce transition on a lexeme while there is no further shift possible on that lexeme. On the other hand, considering a nullable rule as non-nullable will make us refuse transitions that are perfectly viable.

So, we need exactitude here. No sub-optimal scheme would do the job. Performing a global nullability analysis instead of just a local one also provides additional information, such as the detection of cycles and non-decidable grammar rules.

598b ⟨*Computing rule properties, definition* 598b⟩≡
```
METHOD ComputeRuleProperties: BOOLEAN;
```

[4] As a matter of fact, I first implemented such a naive (hence, limited) scheme, thinking it would do the job.

599a ⟨*Computing rule properties, implementation* 599a⟩≡
```
METHOD ComputeRuleProperties: BOOLEAN;
  VAR
    OldLevel, Pass: INTEGER;
    GoOn: BOOLEAN;
  CONST
    Limit = 200;
  BEGIN
  IF DuraCfg.VerboseLevel > 1 THEN
    StdOut.WriteLine (‘‘Computing rule properties”);
    END;
```

By calling `Close` (See page 606), the grammar is closed, in the sense that all the flat rules that are recursively referred to by the root grammar rule are stored in the flat specification. In other words, after that `Close` has been applied, by scanning all the grammar rules in `THIS`, one visits all the rules that can be used when parsing some input that can reduce to the root grammar rule.

599b ⟨*Computing rule properties, implementation* 599a⟩+≡
```
Close;
```

Then, we reset all the grammar rules properties to a default value . . .

599c ⟨*Computing rule properties, implementation* 599a⟩+≡
```
FOR Rule ∈ THIS DO
  Rule.SetNullableLevel (FlatRule.CodeUnknownLevel);
  Rule.SetReducible(FALSE);
  END;
```

Finally, we compute all rules properties, until all rules properties have been computed, or until we have performed `Limit` iterations. `Limit` has been sized (200) in such a way that it is reasonable to claim that if a rule's property has not been determined so far, we will not be able to compute it by further computations.

599d ⟨*Computing rule properties, implementation* 599a⟩+≡
```
GoOn ← TRUE;
WHILE GoOn ∧ (Pass < Limit) DO
  GoOn ← FALSE;
  Pass ← Pass + 1;
  FOR Rule ∈ THIS DO
    ⟨Consider Rule's properties 600a⟩
    GoOn ← GoOn ∨
           (Rule.NullableLevel = FlatRule.CodeUnknownLevel);
    END;
  END;
```

If a rule's nullable level is not determined yet, let's try to recompute it given the current state of our knowledge – understand: given what we know of the nullability of our other grammar rules at this stage (See method *DuraFlat.FlatRule.ComputeNullableLevel* on page 488).

600a ⟨*Consider* Rule*'s properties* 600a⟩≡

```
IF Rule.NullableLevel < FlatRule.CodeNotNullableLevel THEN
  OldLevel ← Rule.NullableLevel;
  Rule.ComputeNullableLevel;
  GoOn ← GoOn ∨ (OldLevel ≠ Rule.NullableLevel);
  END;
```

Then, if Rule is known as being not reducible yet, we check whether it can be, given the current state of our knowledge (See method *DuraFlat.FlatRHS.Reducible* on page 516).

600b ⟨*Consider* Rule*'s properties* 600a⟩+≡

```
IF ¬ Rule.Reducible THEN
  IF ∃RHS ∈ Rule • RHS.Reducible THEN
    Rule.SetReducible (TRUE);
    GoOn ← TRUE;
    END;
  END;
```

If we have performed Limit passes, we display the rules for which nullability could not be decided, and rules that cannot be reduced (See method *DuraFlat.FlatRule.Reducible* on page 491), for instance, rules a with a single right hand side of the form $a ::= T\ a$ where T denotes a terminal symbol.

600c ⟨*Computing rule properties, implementation* 599a⟩+≡

```
IF Pass ≥ Limit THEN
  RESULT ← FALSE;
  FOR Rule ∈ THIS | Rule.NullableLevel = FlatRule.CodeUnknownLevel DO
    StdOut.WriteLine (“Undecidable nullability rule”);
    Rule.Dump(StdOut);
    END;
 ELSE
  RESULT ← TRUE;
  FOR Rule ∈ THIS | ¬ Rule.Reducible DO
    RESULT ← FALSE;
    StdOut.WriteLine (“Unreducible rule”);
    Rule.Dump(StdOut);
    END;
  END;
END ComputeRuleProperties;
```

7.14.7 Computing production sets

For every flattened rule r, the attached production set is defined as all the flattened rules f such that $r \overset{*}{\Rightarrow} f$. It is defined extensively in the ObjectZ specification for `FlatRule` (See page 477).

These production sets are useful for several purposes. First, a situation where a grammar includes a rule r which is included in its own production set qualifies a grammatical cycle. Second, they also help considerably in the log file when one tries to decode a large grammar where numerous intermediate grammar rules clutter the overall structure.

Computing these production sets requires reliable information about which rules are nullable. The algorithm is amazingly similar to the one described above for nullability and reducibility. We simply make full passes on the entire grammar, recomputing all P_rs based on the information at hand, until no P_r set can be changed anymore.

`ComputeProductionSets` returns `TRUE` when computing the production sets has not lead to the detection of any grammatical cycle, and `FALSE` otherwise.

601a ⟨*Computing production sets, definition* 601a⟩≡

```
METHOD ComputeProductionSets: BOOLEAN;
```

601b ⟨*Computing production sets, implementation* 601b⟩≡

```
METHOD ComputeProductionSets: BOOLEAN;
  VAR
    GoOn: BOOLEAN;
    OldCard: INTEGER;
    NTSet: NonTerminalSet;
  BEGIN
  RESULT ← TRUE;
  GoOn ← TRUE;
  WHILE GoOn DO
    GoOn ← FALSE;
    FOR Rule ∈ THIS DO
      ⟨Compute the production set attached to Rule 601c⟩
      END;
    END;
  ⟨Dump the cycles 602b⟩
  END ComputeProductionSets;
```

The production set attached to the rule must be created here. The rule cannot create it by itself because it does not know how many elements the set might contain.

601c ⟨*Compute the production set attached to* `Rule` 601c⟩≡

```
NTSet ← Rule.ProductionSet;
IF NTSet = Λ THEN
  NTSet.CREATE(THIS);
  Rule.SetProductionSet (NTSet);
  END;
```

We call `ComputeProductionSet` (See page 507) to let the rule recompute its own production set, and if the number of elements in its production set evolves, we set `GoOn` to `TRUE` to start an additional iteration.

602a ⟨*Compute the P_r set attached to* `Rule` 602a⟩≡
```
  OldCard ← NTSet.UsedBits;
  Rule.ComputeProductionSet;
  GoOn ← GoOn ∨ (OldCard < NTSet.UsedBits);
```

602b ⟨*Dump the cycles* 602b⟩≡
```
  FOR Rule ∈ THIS |
         Rule.ProductionSet_{Rule.Nr} DO
    RESULT ← FALSE;
    StdOut.WriteLine (“Grammatical cycle”);
    Rule.Dump(StdOut);
    END;
```

7.14.8 Computing $FIRST$, $LAST$, $MIDDLE$, $PRECEDE$ and $FOLLOW$ sets

`ComputeAttachedSets` computes the *FIRST*, *LAST*, *MIDDLE*, *PRECEDE* and *FOLLOW* sets for all the grammar rules in this specification. I guess there might be smarter ways of doing this, but I truly believe it would not be worth the trouble, so I implemented a brute force way.

I first initialize all these sets to empty sets for all grammar rules in the specification, and then iterate until no more elements are added to any of these sets. Each iteration simply recomputes the sets based on terminals, nullability, and *PRECEDE* and *FOLLOW* sets that have been computed on the previous pass.

602c ⟨*Computing* FIRST *and* FOLLOW *sets, definition* 602c⟩≡
```
  METHOD ComputeAttachedSets;
```

All these sets are sets of terminal symbols, implemented by the `TerminalSet` class (See page 558).

603a ⟨*Computing* FIRST *and* FOLLOW *sets, implementation* 603a⟩≡

```
METHOD ComputeAttachedSets;
  VAR
    OldCount, BitCount: INTEGER;
    TheSet: TerminalSet;
  BEGIN
  ⟨First, initialize all attached sets to empty sets 603b⟩
  OldCount ← -1;
  WHILE OldCount ≠ BitCount DO
    OldCount ← BitCount;
    ⟨A single iteration 603c⟩
    ⟨Compute the new BitCount 604a⟩
    END;
  END ComputeAttachedSets;
```

The only reason why the `DuraFlatSpecification` must create these sets rather than letting the `FlatRule` instances create them on the fly is because the specification can easily determine the largest code attached to a lexeme, and create the empty sets accordingly.

603b ⟨*First, initialize all attached sets to empty sets* 603b⟩≡

```
FOR Rule ∈ THIS DO
  TheSet.CREATE;
  Rule.SetPrecedeSet (TheSet);
  TheSet.CREATE;
  Rule.SetFollowSet (TheSet);
  TheSet.CREATE;
  Rule.SetFirstSet (TheSet);
  TheSet.CREATE;
  Rule.SetLastSet (TheSet);
  TheSet.CREATE;
  Rule.SetMiddleSet (TheSet);
  END;
```

Divide and conquer: let's leave the responsibility of actually computing these sets to the rules themselves, given the fact that they have direct access to their entire internal structure (See method *DuraFlat.FlatRule.ComputeAttachedSets* on page 506).

603c ⟨*A single iteration* 603c⟩≡

```
FOR Rule ∈ THIS DO
  Rule.ComputeAttachedSets;
  END;
```

We compute `BitCount` as the number of bits in all the sets of all the rules in the grammar. We go on recomputing the sets attached to the grammar rules as long as new elements are added, that is, as long as `BitCount` evolves.

604a ⟨*Compute the new* `BitCount` 604a⟩≡
```
BitCount ← 0;
FOR Rule ∈ THIS DO
  BitCount ← BitCount + Rule.FirstSet.UsedBits
                      + Rule.LastSet.UsedBits
                      + Rule.PrecedeSet.UsedBits
                      + Rule.FollowSet.UsedBits
                      + Rule.MiddleSet.UsedBits ;
  END;
```

7.14.9 Root rule handling

604b ⟨*Root rule handling, definition* 604b⟩≡
```
METHOD SetRoot (Root: FlatRule);
METHOD Root: FlatRule;
  POST
    NO_CHANGE;
```

604c ⟨*Root rule handling, implementation* 604c⟩≡
```
VAR
  TheRoot: FlatRule;

METHOD Root: FlatRule;
  BEGIN
  RESULT ← TheRoot;
  END Root;
```

SetRoot indicates which rule must be used as root – that is, target production of our entire grammar. DURA will then create an additional rule which sole purpose will be to encapsulate this root grammar rule in a normalized scheme. Stated otherwise, we want to make sure that the grammar rule that reduces the target for the grammar is always of the same form.

605a ⟨*Root rule handling, implementation* 604c⟩+≡

```
METHOD SetRoot (Root: FlatRule);
  VAR
    RHS: FlatRHS;
    TrueRoot: FlatRule;
    NTComp: FlatNonTerminal;
    TComp: FlatTerminal;
    FRedu: FinalReduction;
  BEGIN
  ASSERT TheRoot = Λ;
  ASSERT Root ≠ Λ;
  IF ¬ Includes(Root) THEN
    Insert (0, Root);
    END;
  ⟨Build the genuine root grammar rule TrueRule 605b⟩
  Insert(0, TrueRoot);
  TheRoot ← TrueRoot;
  END SetRoot;
```

DURA automatically defines a new rule (referred to as TrueRoot in this chunk) which is made of a reference to the rule meant to be used as root grammar rule, followed by the predefined EOF (See page 185) token to make sure that no partial matches will be accepted. The target production of our grammar will only be reduced after that this EOF symbol has been encountered.

We also attach a newly created instance of FinalReduction (See page 837) to this normalized grammar rule. There should not be any other such reduction in the grammar.

605b ⟨*Build the genuine root grammar rule* TrueRule 605b⟩≡

```
TrueRoot.CREATE;
RHS.CREATE;
NTComp.CREATE;
NTComp.SetRule (Root);
RHS.Append (NTComp);
TComp.CREATE(YLexicalAnalyzer.EOF);
RHS.Append (TComp);
FRedu.CREATE;
RHS.SetReduction (FRedu);
TrueRoot.Append (RHS);
```

7.14.10 Managing grammar membership

606a ⟨*Closure and rule inclusion handling, definition* 606a⟩≡
```
METHOD Includes (Rule: FlatRule): BOOLEAN;
  POST
    NO_CHANGE;

METHOD Close;
```

606b ⟨*Closure and rule inclusion handling, implementation* 606b⟩≡
```
METHOD Includes (Rule: FlatRule): BOOLEAN;
  BEGIN
  RESULT ← Index(Rule) ≥ 0;
  END Includes;
```

The `Close` method listed hereunder adds all non-terminals that can be reached starting at the non-terminals which are currently in the grammar.

606c ⟨*Closure and rule inclusion handling, implementation* 606b⟩+≡
```
METHOD Close;
  VAR
    Idx: INTEGER;
    TheRule: FlatRule;
  BEGIN
  IF DuraCfg.VerboseLevel > 1 THEN
    StdOut.WriteLine (“Closing the flat specification”);
    END;
  IF (Root ≠ Λ) ∧ ¬ Includes(Root) THEN
    Append (Root);
    ASSERT Includes(Root);
    END;
  ⟨Loop on elements in THIS 606d⟩
  Renumber;
  END Close;
```

This loop is written as a `WHILE` loop, as opposed to a `FOR` loop or to a loop directly on the collection because news elements can be added while processing the loop's body, and a `FOR` loop precomputes the set of indexes it will go through before the first iteration. The purpose here is to have this loop go through newly added elements in a single pass.

606d ⟨*Loop on elements in* `THIS` 606d⟩≡
```
WHILE Idx < Size DO
  TheRule ← THIS_Idx;
  ⟨Consider all reachable rules from TheRule 607a⟩
  Idx ← Idx + 1;
  END;
```

Finding all the rules that can be reached from `TheRule` is merely a question of going through all `TheRule`'s right hand sides, through all components in each of these right hand sides, consider `FlatNonTerminals` (See page 554) only, and ensure the rule they refer to is included in `THIS`.

607a ⟨*Consider all reachable rules from* `TheRule` 607a⟩≡
```
FOR RHS ∈ TheRule DO
  FOR El ∈ RHS DO
    WHAT El OF
      IN FlatNonTerminal:
        IF ¬ Includes (TAG.Rule) THEN
          Append (TAG.Rule);
          END;
        END;
     ELSE
      END;
    END;
  END;
```

607b ⟨`FinalItem` *method definition* 607b⟩≡
```
METHOD FinalItem: FlatItem;
  PRE
    First = Root,
    First.Size = 1,
    First.First.Size = 2;
  POST
    RESULT = First.First.GetItem(2);
```

`FinalItem` returns the item that denotes the end of the parse process. It is the rightmost item in the root rule of the grammar. By construction (See page 605), this grammar rule has a single right-hand side, which must be made of a reference to a non-terminal followed by the `EOF` (See page 185) token. This RHS's length must thus be 2, and the rightmost item has index 2, since a RHS with n components has $n + 1$ items, numbered from 0 to n.

607c ⟨`FinalItem` *method implementation* 607c⟩≡
```
METHOD FinalItem: FlatItem;
  BEGIN
  RESULT ← First.First.GetItem(2);
  END FinalItem;
```

Table 7.1: Merging equivalent grammar rules, results

	Rules before merging	Rules after merging	Gain
COBOL	600	448	25%
C++	328	265	19%
Natural	216	186	14%
RCScript	68	58	15%
Java	137	110	20%

7.14.11 Merging equivalent grammar rules

Merging equivalent grammar rules is a very profitable business: it commonly shaves 25% off from the total amounts of grammar rules in the flattened representation. This can sound as an extreme figure, but the beloved reader should remember that this flattened representation is generated automatically, based on the original grammatical expression-based representation (See page 681). For instance, in `COBOL`, commas can be used as white spaces. One can decide to consider them as white spaces as far as at the lexical level. One can also accept them as part of the entire grammar but mark them as optional, as in:

```
GRAMMAR
  ExprList := Expression + BY Comma? ;
```

It is intuitively clear that such `Comma?` expressions will occur in several places in the grammar, in a form similar to:

$$\begin{aligned} Synth \; ::= \; & Comma \mid \\ & \epsilon \end{aligned}$$

up to a point where it gets highly profitable to merge all these grammar rules into a single, reused grammar rule. This is what the following chunks are all about: detecting the cases where such a replacement can be performed.

608 ⟨*Merging equivalent rules, definition* 608⟩≡

```
  METHOD MergeEquivalentRules;
```

609a ⟨*Merging equivalent rules, implementation* 609a⟩≡

```
METHOD MergeEquivalentRules;
  VAR
    ToReplace: ARRAY OF INTEGER;
    OldSize, OldGain, Gain, FirstIndex, LastIndex: INTEGER;

  ⟨FindEndEquivClass local method 611⟩
  ⟨HandleSlice local method 612a⟩

  BEGIN
  OldGain ← -1;
  OldSize ← Size;
  ToReplace.CREATE (Size);
  ⟨Ensure transitivity 609b⟩
  WHILE OldGain ≠ Gain DO
    OldGain ← Gain;
    ⟨One merging pass 609c⟩
    END;
  ASSERT Size + Gain = OldSize;
  END MergeEquivalentRules;
```

In order to ensure that sorting can happen flawlessly, we first ensure that the comparison function (implemented by the `ShouldSwap` method, as defined originally by the `AbstractList` class, and redefined by `FlatSpecification`) is transitive, that is: $a < b \land b < c \Rightarrow a < c$.

After that this method invocation has allowed me to find the exact cases where comparison was not transitive, I removed it, since it takes over 15' of CPU time when applied to a grammar such as ObjectZ.

609b ⟨*Ensure transitivity* 609b⟩≡

```
DEBUG
  -- EnsureTransitivity;
  END;
```

`ToReplace` holds a translation table, that will be applied onto the entire grammar to replace the references to a given grammar rule by a reference to another one.

609c ⟨*One merging pass* 609c⟩≡

```
FOR i ∈ 0 .. ToReplace.SIZE - 1 DO
  ToReplace_i ← -1;
  END;
```

We sort and renumber the grammar, so that rules that can potentially be merged are close to each other in the list. "Close to each other" is kind of vague, but that's on purpose. Comparing two `FlatRule` instances for possible merge is not such an easy matter (See method *DuraFlat.FlatRule.Compare* on page 509).

609d ⟨*One merging pass* 609c⟩+≡

```
Sort;
Renumber;
```

Using the knowledge we have about the fact that `THIS` is sorted, we consider equivalent classes based on our strange-looking pre-order, and perform a combinatorial search for equivalent rules within such equivalence classes.

610a ⟨*One merging pass* 609c⟩+≡

```
FirstIndex ← 1; -- Skip rule nr 0, which is the root...
WHILE FirstIndex < Size DO
  LastIndex ← FindEndEquivClass (FirstIndex);
  IF LastIndex ≠ FirstIndex THEN
    HandleSlice (FirstIndex, LastIndex);
    END;
  FirstIndex ← LastIndex + 1;
  END;
```

After that all buckets have been visited, if at least a single merge has been done, we renumber the references using the correpondance table `ToReplace`, and remove the rules that are not referred to anymore.

610b ⟨*One merging pass* 609c⟩+≡

```
IF OldGain ≠ Gain THEN
  ⟨Renumber rule references using ToReplace 610c⟩
  ⟨Remove unused rules 610d⟩
  ⟨Dump something if verbose flag is set 610e⟩
  END;
ASSERT Size + Gain = OldSize;
```

(See method *DuraFlSpec.FlatSpecification.RenumberRuleReferences* on page 613).

610c ⟨*Renumber rule references using* `ToReplace` 610c⟩≡

```
RenumberRuleReferences (ToReplace);
```

Removing unused rules happens essentially by removing all the rules for which `ToReplace` indicates that references to them have been replaced by references to another rule.

610d ⟨*Remove unused rules* 610d⟩≡

```
FOR i ∈ Size - 1 .. 0 BY -1 | ToReplace_i ≥ 0 DO
  Delete(i);
  END;
Renumber;
```

610e ⟨*Dump something if verbose flag is set* 610e⟩≡

```
IF DuraCfg.VerboseLevel > 0 THEN
  StdOut.WriteInt (Gain, 0);
  StdOut.WriteString (" out of ");
  StdOut.WriteInt (OldSize, 0);
  StdOut.WriteLine (" rules have been optimized");
  END;
```

`FindEndEquivClass` returns the index within `THIS` that ends an equivalence class starting at index `From`. This method relies on the fact that `THIS` is sorted.

611 ⟨`FindEndEquivClass` *local method* 611⟩≡

```
METHOD FindEndEquivClass (From: INTEGER): INTEGER;
  VAR
    FirstEl: FlatRule;
    Cmp: INTEGER;
    GoOn: BOOLEAN;
  BEGIN
  FirstEl ← THIS_From;
  RESULT ← From;
  GoOn ← TRUE;
  WHILE GoOn DO
    IF RESULT = Size-1 THEN
      GoOn ← FALSE;
     ELSE
      Cmp ← FirstEl.Compare(THIS.Get(RESULT + 1));
      IF (Cmp = FirstEl.Equal) ∨
          (Cmp = FirstEl.AlmostEqual) THEN
        RESULT ← RESULT + 1;
       ELSE
        GoOn ← FALSE;
        END;
      END;
    END;
  END FindEndEquivClass;
```

`HandleSlice` handles a slice of `THIS`, in the sense that it checks whether some of the rules within the slice can be merged. `THIS` was first sorted according to a preorder, to ensure that such quadratic processes are only performed on slices of limited size.

612a ⟨`HandleSlice` *local method* 612a⟩≡

```
METHOD HandleSlice (FromIndex, ToIndex: INTEGER);
  VAR
    Visited: ARRAY OF BOOLEAN;
    RefEl: FlatRule;
  BEGIN
  Visited.CREATE (ToIndex-FromIndex+1);
  FOR i ∈ 0 .. Visited.SIZE - 1 | ¬ Visited_i DO
    RefEl ← THIS_{i+FromIndex};
    FOR j ∈ i+1 .. Visited.SIZE - 1 | ¬ Visited_j DO
      IF (RefEl.Compare(THIS_{FromIndex+j} = RefEl.Equal) THEN
        ToReplace_{FromIndex+j} ← FromIndex + i;
        Visited_j ← TRUE;
        Gain ← Gain + 1;
        END;
      END;
    END;
  END HandleSlice;
```

612b ⟨*Ensure antisymmetry* 612b⟩≡

```
DEBUG
  IF RESULT ∧ (Two.Compare (One) = One.Greater) THEN
    One.Dump (StdOut);
    Two.Dump (StdOut);
    ASSERT FALSE;
    END;
  END;
```

YAFL

The comparison method which is used to sort a list made of `FlatRule` instances will only swap elements if comparing them yields `Greater`. For the scope of this sorting method, quasi-equality (as represented by the value `AlmostEqual`) is considered as plain equality `Equal` (See page 509).

612c ⟨*Merging equivalent rules, implementation* 609a⟩+≡

```
REDEFINE METHOD ShouldSwap (One, Two: FlatRule): BOOLEAN;
  BEGIN
  RESULT ← (One.Compare(Two) = One.Greater);
  ⟨Ensure antisymmetry 612b⟩
  END ShouldSwap;
```

The sorting method is redefined, in order to preserve the position of the root grammar rule, which should always be the first one in the list (See method *DuraFlSpec.FlatSpecification.SetRoot* on page 605). The root grammar rule is removed, the rest of the list is sorted by calling `BASE`, and the root rule is inserted back again.

613a ⟨*Merging equivalent rules, implementation* 609a⟩+≡
```
REDEFINE METHOD Sort;
  VAR
    El: FlatRule;
  BEGIN
  ASSERT Index(Root)=0;
  El ← First;
  Delete(0);
  BASE;
  Insert(0, El);
  END Sort;
```

Rules and right hand sides must be numbered independently to be able to refer to them using integers, that can be stored in tables for later retrieval, far more easily than references.

613b ⟨*Numbering methods* 613b⟩≡
```
METHOD Renumber;
  VAR
    RhsNr: INTEGER;
    FRule: FlatRule;
  BEGIN
  FOR i ∈ 0 .. Size - 1 DO
    FRule ← THIS_i;
    FRule.SetNr (i);
    FOR Rhs ∈ FRule DO
      Rhs.SetNr(RhsNr);
      RhsNr ← RhsNr + 1;
      END;
    END;
  END Renumber;
```

The `RenumberRuleReferences` method takes a mapping array as parameter, and patches all the rules in the specifications according to this array. This method is used when merging equivalent grammar rules (See method *DuraFlSpec.FlatSpecification.MergeEquivalentRules* on page 608), so that all the references to rules in the equivalence class are replaced by references to the rule that *represents* the equivalence class.

By replacing references, this method implicitly makes some rules unreachable. Such useless rules are removed by a separate process (See method *DuraFlSpec.FlatSpecification.RemoveUnreachableRules* on page 565).

614a ⟨*Numbering methods* 613b⟩+≡
```
METHOD RenumberRuleReferences (Map: ARRAY OF INTEGER);
  VAR
    NewNr: INTEGER;
  BEGIN
  FOR i ∈ 0 .. Size-1 | Map_i < 0 DO
    FOR RHS ∈ THIS_i DO
      FOR El ∈ RHS DO
        WHAT El OF
          IN FlatNonTerminal:
            NewNr ← Map_TAG.Rule.Nr;
            IF NewNr ≥ 0 THEN
              TAG.SetRule (THIS_NewNr;
              END;
            END;
         ELSE
          END;
        END;
      END;
    END;
  END RenumberRuleReferences;
```

614b ⟨*Error handling, definition* 614b⟩≡
```
METHOD SetError (Fl: BOOLEAN);
METHOD Error: BOOLEAN;
```

614c ⟨*Error handling, implementation* 614c⟩≡
```
VAR
  TheErr: BOOLEAN;

METHOD SetError (Fl: BOOLEAN);
  BEGIN
  TheErr ← Fl;
  END SetError;

METHOD Error: BOOLEAN;
  BEGIN
  RESULT ← TheErr;
  END Error;
```

614d ⟨LastRHSNr *method definition* 614d⟩≡
```
METHOD LastRHSNr: INTEGER;
  POST
    RESULT ≥ 0;
```

LastRHSNr returns the number of the last right hand side in the grammar, which is required to size tables that implement functions that operate on right hand sides (See *DuraFlSpec.FlatSpecification.TheLengthTable* on page 597).

615 ⟨LastRHSNr *method implementation* 615⟩≡

```
METHOD LastRHSNr: INTEGER;
  VAR
    Rule: FlatRule;
  BEGIN
  Rule ← LAST r IN THIS • r.Size > 0;
  RESULT ← Rule.Last.Nr;
  END LastRHSNr;
```

7.14.12 Ensuring transitivity of the comparison function

The following chunk has been added when replacing linked lists by array-based lists with the same interface. Why, might a naive reader ask ? What is the connection between the way lists are implemented and an algebraic property such as the transitivity of the comparison function ?

Well, beloved reader, there is such a connection. When sorting linked lists, we build sorted sublists which are merged afterwards. Such an algorithm is specially suited to lists, since it can be implemented without ever reallocating an element; it provides worst case performance similar to quicksort's ideal case, and it is stable – that is, it does not swap two elements unless they are out of order. Unfortunately, such an algorithm behaves quite poorly when applied to arrays. Hence, when switching from linked lists to array-based lists, we also replaced the sorting algorithm by a slightly improved implementation of the quicksort ([85] is not the first reference, it is not the only reference either, but to some extend, it can be considered to be for algorithms what is for compilers).

The true difference between the two algorithms, beyond marginal performance issues that have had little impact on practical usability, is the sensitivity to ill-formed order or pre-order relationships. When sorting lists, one basically compares two elements at a time, and considers the fact that they are ordered or not. In the undesired case where the comparison function would *never* consider them as sorted, the system swaps them once.

On the other hand, given two elements a and b for which $a < b$ and $b < a$ (such a "$<$" relationship is of course not a preorder), some quicksort implementations might cycle. More specifically, it is not quicksort itself that might cycle, but rather, the alternate algorithm which is selected when the number of elements to sort is smaller than a given limit. In our case, the offending algorithm which is used to sort less than 5 elements is a bubble sort.

When sorting a `FlatSpecification`, I soon discovered that my comparison function was ill-formed, since it caused the bubble sort, and consequently, the enclosing quicksort to cycle. I first changed my comparison function $<$ to be antisymmetric, and I ensured that it is indeed the case whenever I compare two elements. A similar check has been included in the `ArrayList` class as well, to prevent a similar problem to occur without being noticed in another context.

Antisymmetry is easy to ensure, since it relies on two elements. Whenever I compare two distinct elements a and b, I can ensure that $a < b \Leftrightarrow b \not< a$ or at least, $a < b \Rightarrow b \not< a$.

However, our problems with quicksort did not stop here. When considering swapping an element l in the left slice such as $l > p$, p being the pivot, and an element r in the right slice, such as as $r < p$, we implicitely use the transitivity property of our preorder relationship, and from $l > p > r$, we deduce $l > r$.

The algorithm crashed on one occurence, because my order relationship was not transitive. In the case of our list-based sorting algorithm, this did not prevent the sorting algorithm from terminating, since by construction, the maximum number of comparison is bounded, no matter how ill-formed the comparison function might be. On the other hand, in some configurations, a quicksort can loop endlessly if the comparison function does not defined e well-formed order relationship.

The following chunk implements transitivity checking. Checking for transitivity if far more painful than antisymmetry, since it requires three elements instead of two, hence has cubic complexity while antisymmetry was only quadratic.

`EnsureTransitivity` relies on `ShouldSwap` as redefined by `FlatSpecification` (See page 612) to compare elements. For the rest, I would not qualify this as truly exciting code.

617 ⟨*Ensuring transitivty of the comparison function* 617⟩≡

```
METHOD EnsureTransitivity;
  VAR
    One, Two, Three: FlatRule;
  BEGIN
  DEBUG
    Renumber;
    FOR i ∈ 0 .. Size - 1 DO
      IF i MOD 16 = 0 THEN
        StdOut.WriteInt (i, 0);
        StdOut.WriteChar (‘‘/”);
        StdOut.WriteInt (Size, 0);
        StdOut.WriteLn;
        END;
      One ← THIS_i;
      FOR j ∈ 0 .. Size - 1 | i ≠ j DO
        Two ← THIS_j;
        IF ShouldSwap (One, Two) THEN
          FOR k ∈ 0 .. Size - 1 | (i ≠ k) ∧ (j ≠ k) DO
            Three ← THIS_k;
            IF (ShouldSwap (Two, Three) ∧
                ¬ ShouldSwap (One, Three)) THEN
              One.Dump (StdOut);
              Two.Dump (StdOut);
              Three.Dump (StdOut);
              Λ ← ShouldSwap (One, Two);
              Λ ← ShouldSwap (Two, Three);
              Λ ← ShouldSwap (One, Three);
              ASSERT FALSE;
              END;
            END;
          END;
        END;
      END;
    END;
  END EnsureTransitivity;
```

7.14.13 Mundane stuff

Well, not much to say...

618a ⟨*Definition import list* 618a⟩≡
```
FROM DuraFlat IMPORT FlatRule, FlatRHS, TerminalSet, NonTerminalSet;
FROM DuraIntDictionary IMPORT IntDictionary;
FROM DuraItem IMPORT FlatItemSet, FlatItem;
FROM DuraOutput IMPORT DuraOutputStream;
FROM DuraPDA IMPORT PDAState, PushDownAutomaton;
FROM DuraTransitions IMPORT TerminalTransition, NonTerminalTransition,
                            NonTerminalTransitionList,
                            TerminalTransitionList;

IMPORT ArrayList;
FROM Streams IMPORT OutputStream, StdOut;
```

618b ⟨*Implementation import list* 618b⟩≡
```
FROM BitMap IMPORT BitMap;
FROM DuraCfg IMPORT DuraCfg;
FROM DuraFlat IMPORT FlatTerminal, FlatNonTerminal, FlatTerminalSet,
                     FlatRuleComponent;
FROM DuraIntermediateFormat IMPORT InterAttribute, InterType,
                                   InterStringType, InterIntegerType,
                                   InterBooleanType;
FROM DuraPDA IMPORT PDAStateList;
FROM DuraReductions IMPORT FinalReduction, DuraReduction;

FROM Conversions IMPORT IntConversions;
FROM Streams IMPORT StdOut;
IMPORT Set;
IMPORT Stack;
IMPORT SYSTEM;
FROM YLexicalAnalyzer IMPORT YLexicalAnalyzer;
```

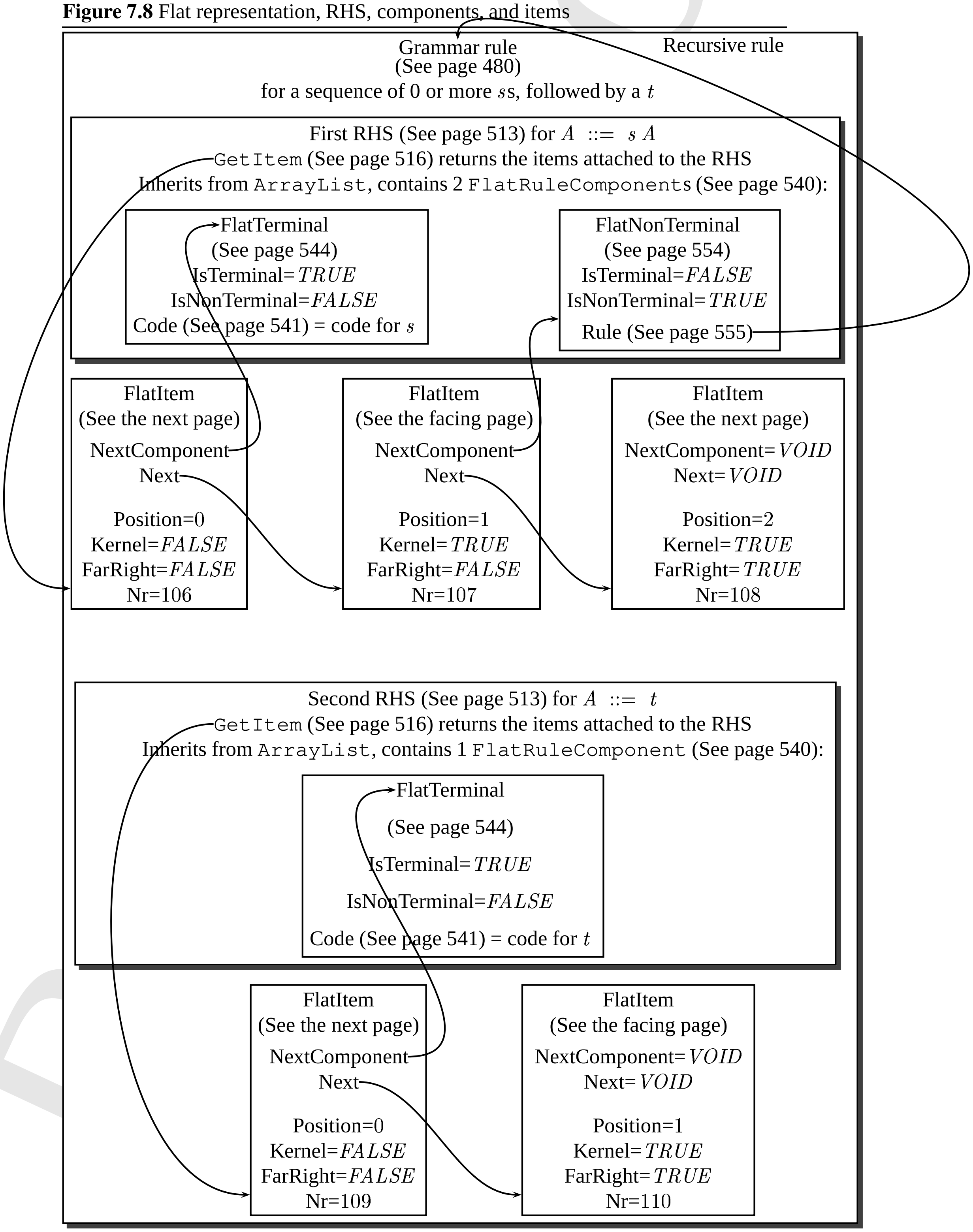
7.15 DuraItem

`DuraItem` defines LR *items*, as described in . Items are *positions* within grammar rules. A right hand side made of n elements defines $n + 1$ items, one *before* every symbol in the right hand side, and one additional one *after* the last symbol in the right hand side. Figure 7.8 depicts the relationships between these items and the other classes anc constructs managed within `DURA` .

⟨*Definition* ⟩≡

```
DEFINITION MODULE DuraItem;

⟨DuraItem definition import list 635a⟩

    ⟨FlatItem class definition 622⟩
    ⟨FlatItemList class definition 629b⟩
    ⟨FlatItemSet class definition 630a⟩

END DuraItem;
```

Figure 7.8 Flat representation, RHS, components, and items



⟨*Implementation* ⟩≡
IMPLEMENTATION MODULE `DuraItem;`

⟨`DuraItem` *implementation import list* 635b⟩

⟨`FlatItem` *class implementation* 624⟩
⟨`FlatItemDictionary` *class implementation* 629d⟩
⟨`FlatItemSet` *class implementation* 630b⟩
⟨`FlatItemList` *class implementation* 629c⟩

END `DuraItem;`

A `FlatItem` is a position within a rule, or more specifically, a position within a RHS (Right hand side) (See class *DuraFlat.FlatRHS* on page 513). It represents the ability for the parser to be in a current position while recognizing a given piece of grammar rule.

7.15.1 FlatItem

`FlatItem`'s are created on demand (See method *DuraFlat.FlatRHS.GetItem* on page 516), and attached to their `RHS`, in such a way that two `FlatItem` instances can be compared for equality using reference equality only.

622 ⟨FlatItem *class definition* 622⟩≡

```
CLASS FlatItem;
  REDEFINE METHOD CREATE(RHS: FlatRHS;
                                  Position: INTEGER);
    PRE
      Position ≥ 0,
      Position ≤ RHS.Size;

  METHOD RHS: FlatRHS;
  METHOD Position: INTEGER;
  METHOD Nr: INTEGER;
  METHOD FarRight: BOOLEAN;
    POST
      RESULT ⇔ Position ≥ RHS.Size;

  METHOD Kernel: BOOLEAN;
    POST
      RESULT ⇔ Position ≠ 0;

  METHOD Next: FlatItem;
    PRE
      ¬ FarRight;

  METHOD NextComponent: FlatRuleComponent;
    PRE
      ¬ FarRight;

  ⟨Closure management, definition 626d⟩
  ⟨Displaying a FlatItem onto an OutputStream, definition 628c⟩

END FlatItem;
```

FlatItem

A $FlatItem$ denotes a position within a $FlatRHS$, referred to by the $Enclosing$ attribute, between two adjacent $FlatRuleComponent$s, before the leftmost one, or after the rightmost one. Since most of its properties will be deduced from the context of its enclosing right hand side, it is no wonder this $Enclosing$ attribute is used whenever something must be known about the item.

$Enclosing : \rho FlatRHS$ (See page 511)
$Position : \mathbb{N}$
$Kernel, FarRight : \mathbb{B}$

$NextComponent$ yields a reference to the $FlatRuleComponent$ (See page 540) which is after the present $FlatItem$. It yields Λ if the $FlatItem$ is at the rightmost position in the right hand side.

$NextComponent : \rho \downarrow FlatRuleComponent$

$\delta Enclosing.Items(Position) = self$
$Kernel = \text{true} \Leftrightarrow Position > 0$
$FarRight = \text{true} \Leftrightarrow Position = \delta Enclosing.Length + 1$
$FarRight = \text{true} \Leftrightarrow NextComponent = \Lambda$
$FarRight = \text{false} \Leftrightarrow$
$\quad \delta NextComponent = \delta Enclosing.Components(Position)$

The following attribute definition only includes very general constraints about $Closure$. More detailed information about the exact content of this $Closure$ set will be defined depending on the $FlatRuleComponent$ (See page 540) referred to by $NextComponent$.

$Closure : \mathbb{P}\, \rho FlatItem$

$Closure$ must contain a reference to $self$

$\exists\, x \in Closure \bullet \delta x = self$

and must be a superset of the closures of all the $FlatItem$s it contains.
$\forall\, x \in Closure \bullet \delta x.Closure \subset Closure$

$Transition$ is the function which, given a $FlatItem$ and a $Symbol$, gives the set of the $FlatItem$s one can go to. It is used to compute the $goto$ function as described extensively in .

$$Transition : Symbol \mapsto \mathbb{P}\, \rho FlatItem$$

$FarRight$ items have no outgoing transitions, since they don't have a symbol on their right on which the transition can then occur.

$$FarRight = \text{true} \Leftrightarrow \forall\, s \in Symbol \bullet Transition(s) = \phi$$

The following property essentially expresses the fact that the set of $FlatItem$s (See the page before) returned by $Transition$ must be closed. Whenever an item is in the set it returns, the entire closure of this item must be in the set as well.

$$\forall\, s \in Symbol, \forall\, x \in Transition(s) \bullet \delta x.Closure \subset Transition(s)$$

624 ⟨`FlatItem` *class implementation* 624⟩≡

```
CLASS FlatItem;
  VAR
    TheNr: INTEGER;
    TheRHS: FlatRHS;
    ThePosition: INTEGER;

  ⟨CREATing a FlatItem 625a⟩
  ⟨Trivial consultation methods 625b⟩
  ⟨Almost trivial consultation methods 625c⟩
  ⟨Displaying a FlatItem onto an OutputStream, implementation 628d⟩
  ⟨Closure management, implementation 626e⟩

END FlatItem;
```

When `CREAT`ing a `FlatItem`, various attributes are set, a unique number is given to `TheNr`, and the item is stored in the dictionary (See page 629). Sets of items will be turned into PDA states (See page 636) using the unique numbering scheme to represent these PDA states as bitmaps, and the dictionary will then be used to retrieve the items that were used to build a PDA state.

625a ⟨`CREAT`*ing a* `FlatItem` 625a⟩≡

```
REDEFINE METHOD CREATE(RHS: FlatRHS;
                          Position: INTEGER);
  BEGIN
  BASE;
  TheNr ← FlatItemDictionary.Size;
  FlatItemDictionary.Append (THIS);
  TheRHS ← RHS;
  ThePosition ← Position;
  END CREATE;
```

Not much to say, I guess...

625b ⟨*Trivial consultation methods* 625b⟩≡

```
METHOD RHS: FlatRHS;
  BEGIN
  RESULT ← TheRHS;
  END RHS;

METHOD Position: INTEGER;
  BEGIN
  RESULT ← ThePosition;
  END Position;

METHOD Nr: INTEGER;
  BEGIN
  RESULT ← TheNr;
  END Nr;
```

`FarRight` merely indicates whether `THIS` refers to the last position within its attached `FlatRHS`, that is, the position *after* the last terminal or non-terminal of the right hand side.

625c ⟨*Almost trivial consultation methods* 625c⟩≡

```
METHOD FarRight: BOOLEAN;
  BEGIN
  RESULT ← ThePosition ≥ RHS.Size;
  END FarRight;
```

626a ⟨*Almost trivial consultation methods* 625c⟩+≡
```
METHOD Kernel: BOOLEAN;
  BEGIN
  RESULT ← ThePosition > 0;
  END Kernel;
```

`Next` returns the next *item* attached to `THIS`'s right hand side, not to be confused with the next `FlatRuleComponent`.

626b ⟨*Almost trivial consultation methods* 625c⟩+≡
```
METHOD Next: FlatItem;
  BEGIN
  ASSERT ¬ FarRight;
  RESULT ← RHS.GetItem(Position+1);
  END Next;
```

Now that you mention it, well, `NextComponent` returns the rule component that lies in the `FlatRHS` after the position referred to by `THIS`.

626c ⟨*Almost trivial consultation methods* 625c⟩+≡
```
METHOD NextComponent: FlatRuleComponent;
  BEGIN
  ASSERT ¬ FarRight;
  RESULT ← TheRHS_Position;
  END NextComponent;
```

Given these basic tools, let's now describe some of the more elaborate operations that can be performed on `FlatItem` instances.

Closures are computed explicitely using `ComputeClosure`, and can then be queried efficiently using `Closure`.

626d ⟨*Closure management, definition* 626d⟩≡
```
METHOD Closure: FlatItemSet;
METHOD ComputeClosure: INTEGER;

METHOD Reachable (Len: INTEGER): FlatItemSet;
```

626e ⟨*Closure management, implementation* 626e⟩≡
```
VAR
  TheClosure: FlatItemSet;

METHOD Closure: FlatItemSet;
  BEGIN
  RESULT ← TheClosure;
  END Closure;
```

627a ⟨*Closure management, implementation* 626e⟩+≡

```
METHOD ComputeClosure: INTEGER;
  BEGIN
  IF TheClosure = Λ THEN
    TheClosure.CREATE;
    END;
  ⟨First, add THIS that can be reached, of course 627b⟩
  IF ¬ FarRight THEN
    ⟨Add whatever can be reached by the component on the right 627c⟩
    END;
  RESULT ← TheClosure.UsedBits;
  END ComputeClosure;
```

It may sound a little bit naive, but starting at `THIS`, `THIS` is reachable without consuming any input terminal or non-terminal.

627b ⟨*First, add* `THIS` *that can be reached, of course* 627b⟩≡

```
TheClosure.AddItem(THIS);
```

If `NextComponent` is a terminal component (See page 544), its closure is empty. If it is a non-terminal component, it will be computed at some time (See page 556).

627c ⟨*Add whatever can be reached by the component on the right* 627c⟩≡

```
TheClosure.Or (NextComponent.Closure);
```

`Reachable` implement a generalization of the closure concept: while the closure of an item is the set of items that can be reached *without* any transition, `Reachable` computes the set of items that can be reached after a number of transitions denoted by `Len`. These transitions can include terminal as well as non-terminal transitions. We are not dealing here with shift or reduces, but merely, with the idea of *advancing* by `Len` symbols (terminals or non-terminals).

Reachable works by first computing the set of items that can be reached after 1 transition into TheReachableOne. If another length than 1 is necessary, the similarly named method for FlatItemSet (See page 634) is called recursively on TheReachableOne with a reduced length.

628a ⟨*Closure management, implementation* 626e⟩+≡

```
VAR
  TheReachableOne: FlatItemSet;

METHOD Reachable (Len: INTEGER): FlatItemSet;
  BEGIN
  IF Len = 0 THEN
    RESULT ← Closure;
   ELSE
    IF TheReachableOne = Λ THEN
      ⟨Compute TheReachableOne 628b⟩
      END;
    IF Len > 1 THEN
      RESULT ← TheReachableOne.Reachable(Len - 1);
     ELSE
      RESULT ← TheReachableOne;
      END;
    END;
  END Reachable;
```

628b ⟨*Compute* TheReachableOne 628b⟩≡

```
TheReachableOne.CREATE;
FOR Item ∈ Closure.BuildList | ¬ Item.FarRight DO
  ASSERT Item.Next ≠ Λ;
  TheReachableOne.AddItem (Item.Next);
  END;
TheReachableOne ← TheReachableOne.Closure;
```

628c ⟨*Displaying a* FlatItem *onto an* OutputStream*, definition* 628c⟩≡

```
METHOD Dump (Output: OutputStream);
```

628d ⟨*Displaying a* FlatItem *onto an* OutputStream*, implementation* 628d⟩≡

```
METHOD Dump (Output: OutputStream);
  BEGIN
  Output.WriteChar (‘‘[”);
  Output.WriteInt (Nr, 0);
  Output.WriteChar (‘‘]”);
  END Dump;
```

629a ⟨*Displaying a* `FlatItem` *onto an* `OutputStream`*, implementation* 628d⟩+≡

```
METHOD DumpWithinRule (Output: OutputStream);
  BEGIN
  RHS.DumpItem(Output, Position);
  END DumpWithinRule;
```

7.15.2 Lists of `FlatItem`

`FlatItemList` implements lists of flat items, where the only added values lies in its ability to sort such strange beasts in a reasonable way.

629b ⟨`FlatItemList` *class definition* 629b⟩≡

```
CLASS FlatItemList;
  INHERITS ArrayList(FlatItem);

END FlatItemList;
```

629c ⟨`FlatItemList` *class implementation* 629c⟩≡

```
CLASS FlatItemList;
  INHERITS ArrayList(FlatItem);

REDEFINE METHOD ShouldSwap (One, Two: FlatItem): BOOLEAN;
  BEGIN
  IF ¬ One.FarRight THEN
    IF Two.FarRight THEN
      RESULT ← TRUE;
     ELSE
      RESULT ← One.NextComponent.Code >
                         Two.NextComponent.Code;
      END;
    END;
  END ShouldSwap;

END FlatItemList;
```

`FlatItemDictionary` keeps track of all items defined within a run of `DURA`, so that we can use the induced implicit mapping from integers to `FlatItem`s to implement sets of `FlatItem`s efficiently as bitmaps, where the n^{th} bit is set to indicate that the set contains the n^{th} element of the `FlatItemDictionary`.

629d ⟨`FlatItemDictionary` *class implementation* 629d⟩≡

```
ONCE CLASS FlatItemDictionary;
  INHERITS FlatItemList;
END FlatItemDictionary;
```

7.15.3 Sets of `FlatItem`

A `FlatItemSet` is a set of `FlatItem`. Typically it is used to represent the ability for the parser to be in a state that corresponds to any of these items.

`FlatItemSet` inherits from `HashElement`, to support convenient and efficient storage and value-based retrieval in hash tables.

630a ⟨`FlatItemSet` *class definition* 630a⟩≡

```
CLASS FlatItemSet;
  INHERITS HashElement;

  REDEFINE METHOD CREATE;
  ⟨Simple set operations on FlatItemSet, definition 631b⟩
  ⟨Displaying a FlatItemSet on an OutputStream, definition 633b⟩
  ⟨FlatItemSet's closure management, definition 633d⟩
  ⟨Checking whether two FlatItemSet instances are equal, definition 632c⟩

  ⟨BuildList method definition 632f⟩

END FlatItemSet;
```

`FlatItemSet` uses a bitmap (See page 843) where individual bits indicate the presence in the set of the corresponding `DuraItem` in `FlatItemDictionary`.

630b ⟨`FlatItemSet` *class implementation* 630b⟩≡

```
CLASS FlatItemSet;
  INHERITS HashElement;

  VAR
    BitMap: DuraBitMap;

  REDEFINE METHOD CREATE;
    BEGIN
    BASE;
    BitMap.CREATE (FlatItemDictionary.Size);
    END CREATE;

  ⟨Computing a FlatItemSet's hash value, implementation 631a⟩
  ⟨Simple set operations on FlatItemSet, implementation 631c⟩
  ⟨Normalization method 632e⟩
  ⟨Displaying a FlatItemSet on an OutputStream, implementation 633c⟩
  ⟨FlatItemSet's closure management, implementation 634a⟩
  ⟨Checking whether two FlatItemSet instances are equal, implementation 632d⟩
  ⟨BuildList method implementation 633a⟩

END FlatItemSet;
```

We reuse `DuraBitMap`'s hash function (See method *DuraBitMap.DuraBitMap.HashValue* on page 845).

631a ⟨*Computing a* `FlatItemSet`*'s hash value, implementation* 631a⟩≡

```
REDEFINE METHOD HashValue: INTEGER;
  BEGIN
  RESULT ← BitMap.HashValue;
  END HashValue;
```

631b ⟨*Simple set operations on* `FlatItemSet`*, definition* 631b⟩≡

```
METHOD AddItem (Item: FlatItem);
METHOD Intersects (Other: FlatItemSet): BOOLEAN;
METHOD IncludesItem (Item: FlatItem): BOOLEAN;
METHOD Or (Other: FlatItemSet);
METHOD UsedBits: INTEGER;
```

Adding an item to a `FlatItemSet` is merely a question of setting the corresponding bit in `BitMap`

631c ⟨*Simple set operations on* `FlatItemSet`*, implementation* 631c⟩≡

```
METHOD AddItem (Item: FlatItem);
  BEGIN
  ⟨Resetting the caches 634c⟩
  BitMap.Set (Item.Nr);
  END AddItem;
```

Similarly, `FlatItemSet` intersection is based on the corresponding `BitMap`'s intersection.

631d ⟨*Simple set operations on* `FlatItemSet`*, implementation* 631c⟩+≡

```
METHOD Intersects (Other: FlatItemSet): BOOLEAN;
  BEGIN
  IF Other ≠ Λ THEN
    RESULT ← BitMap.Intersects (Other.BitMap);
    END;
  END Intersects;
```

Checking whether a `FlatItemSet` includes a given item is merely a question of testing the corresponding bit in `BitMap`.

631e ⟨*Simple set operations on* `FlatItemSet`*, implementation* 631c⟩+≡

```
METHOD IncludesItem (Item: FlatItem): BOOLEAN;
  BEGIN
  RESULT ← (Item ≠ Λ) ∧ BitMap[Item.Nr];
  END IncludesItem;
```

Here again, `Or`ing two `FlatItemSets` translates into `Or`ing the attached bitmaps. However, since `Or`ing the bitmaps can change the current value of cached information such as the closure, we reset the attributes in which this information is maintained.

632a ⟨*Simple set operations on* `FlatItemSet`*, implementation* 631c⟩+≡

```
METHOD Or (Other: FlatItemSet);
  BEGIN
  IF (Other ≠ Λ) ∧ (Other.BitMap ≠ Λ) THEN
    ⟨Resetting the caches 634c⟩
    BitMap.Or (Other.BitMap);
    END;
  END Or;
```

How much is there to comment here ?

632b ⟨*Simple set operations on* `FlatItemSet`*, implementation* 631c⟩+≡

```
METHOD UsedBits: INTEGER;
  BEGIN
  RESULT ← BitMap.UsedBits;
  END UsedBits;
```

632c ⟨*Checking whether two* `FlatItemSet` *instances are equal, definition* 632c⟩≡

```
METHOD IsEqual (Other: FlatItemSet): BOOLEAN;
```

This is getting better and better . . .

632d ⟨*Checking whether two* `FlatItemSet` *instances are equal, implementation* 632d⟩≡

```
METHOD IsEqual (Other: FlatItemSet): BOOLEAN;
  BEGIN
  RESULT ← BitMap.IsEqual (Other.BitMap);
  END IsEqual;
```

Normalizing a bitmap means using the bitmap page dictionary to reuse already allocated bitmaps fragments whenever possible (See method *DuraBitMap.DuraBitMap. Normalize* on page 845).

632e ⟨*Normalization method* 632e⟩≡

```
METHOD Normalize;
  BEGIN
  BitMap.Normalize;
  END Normalize;
```

632f ⟨`BuildList` *method definition* 632f⟩≡

```
METHOD BuildList: FlatItemList;
```

`BuildList` returns the elements in the `FlatItemSet` it is being applied to, by extracting the elements matching the bits set in `BitMap` from the dictionary `FlatItem-Dictionary`. It stores the result in a cache attribute `List` so that further invocations do not require the reallocation of a new list.

633a ⟨`BuildList` *method implementation* 633a⟩≡

```
VAR
  TheList: FlatItemList;

METHOD BuildList: FlatItemList;
  BEGIN
  IF TheList = Λ THEN
    TheList.CREATE;
    FOR i ∈ 0 .. FlatItemDictionary.Size - 1 |
        BitMap_i DO
      TheList.Append(FlatItemDictionary_i;
      END;
    END;
  RESULT ← TheList;
  END BuildList;
```

633b ⟨*Displaying a* `FlatItemSet` *on an* `OutputStream`*, definition* 633b⟩≡

```
METHOD Dump (Output: OutputStream);
```

633c ⟨*Displaying a* `FlatItemSet` *on an* `OutputStream`*, implementation* 633c⟩≡

```
METHOD Dump (Output: OutputStream);
  BEGIN
  FOR i ∈ 0 .. BitMap.Size - 1 |
       (BitMap_i ∧ FlatItemDictionary_i.Kernel DO
    FlatItemDictionary_i;
    END;
  Output.WriteLn;
  END Dump;
```

633d ⟨`FlatItemSet`*'s closure management, definition* 633d⟩≡

```
METHOD Closure: FlatItemSet;
METHOD Reachable(Len: INTEGER): FlatItemSet;
```

634a ⟨FlatItemSet*'s closure management, implementation* 634a⟩≡

```
VAR
  TheClosure: FlatItemSet;

METHOD Closure: FlatItemSet;
  BEGIN
  IF TheClosure = Λ THEN
    TheClosure.CREATE;
    FOR Item ∈ BuildList DO
      TheClosure.Or (Item.Closure);
      END;
    ⟨Normalize TheClosure 634b⟩
    END;
  RESULT ← TheClosure;
  END Closure;
```

Normalizing TheClosure means optimizing its storage and allowing reference-based comparisons for bitmaps (See method *DuraItem.FlatItemSet.Normalize* on page 632).

634b ⟨*Normalize* TheClosure 634b⟩≡

```
TheClosure.Normalize;
```

TheClosure and TheList are attributes that hold the last values returned by methods (See methods *DuraItem.FlatItemSet.Closure* on the current page and *DuraItem.FlatItemSet.BuildList* on the preceding page) in order to improve performance. When altering the object, these cached attributes must sometimes be reset.

634c ⟨*Resetting the caches* 634c⟩≡

```
TheClosure ← Λ;
TheList ← Λ;
```

Reachable implements a generalization of the closure: while the closure is the set of the items that can be reached *without* a transition, Reachable returns the set of items that can be reached after a number Len of transitions. It is based on a similar method for items (See method *DuraItem.FlatItem.Reachable* on page 627).

634d ⟨FlatItemSet*'s closure management, implementation* 634a⟩+≡

```
METHOD Reachable(Len: INTEGER): FlatItemSet;
  BEGIN
  RESULT.CREATE;
  FOR Item ∈ BuildList DO
    RESULT.Or (Item.Reachable(Len));
    END;
  END Reachable;
```

635a ⟨DuraItem *definition import list* 635a⟩≡
```
IMPORT ArrayList;
FROM DuraFlat IMPORT FlatRHS, FlatRuleComponent;
FROM DuraTransitions IMPORT TerminalTransition, NonTerminalTransition,
                             NonTerminalTransitionList;
FROM Streams IMPORT StdOut, OutputStream;
IMPORT HashElement;
```

635b ⟨DuraItem *implementation import list* 635b⟩≡
```
FROM DuraBitMap IMPORT DuraBitMap;
FROM DuraIntermediateFormat IMPORT InterRule;
FROM DuraFlat IMPORT FlatTerminal, FlatNonTerminal;
FROM DuraReductions IMPORT FetchNonTerminalReduction;

IMPORT HashSpace;
IMPORT Set;
IMPORT Stack;
```

7.16 DuraPDA

This module implements the LR automaton, by defining several classes:

`PDAState` (See this page) which implements a state in such an automaton

`PDAStateList` (See page 650) which implements a collection of such states, providing a comparison method for sorting

`PushDownAutomaton` (See page 651) which implements the actual automaton, providing services such as various queries, statistics and code generation.

⟨*Definition* ⟩≡
DEFINITION MODULE `DuraPDA;`

⟨`DuraPDA` *definition import list* 658a⟩

⟨`PDAState` *class definition* 637⟩
⟨`PDAStateList` *class definition* 650c⟩
⟨`PushDownAutomaton` *class definition* 651⟩

END `DuraPDA;`

⟨*Implementation* ⟩≡
IMPLEMENTATION MODULE `DuraPDA;`

⟨`DuraPDA` *implementation import list* 658b⟩

⟨`PDAState` *class implementation* 639a⟩
⟨`PDAStateList` *class implementation* 650d⟩
⟨`PushDownAutomaton` *class implementation* 652⟩

END `DuraPDA;`

7.16.1 `PDAState`

As described in , a state in a LR automaton is made of a set of LR items, implemented here by the `FlatItemSet` (See page 630) class. Such a set of items must be given when invoking `CREATE`, and will define the `PDAState` in term of its value. In other words, two `PDAState`'s will be considered equal if and only if their attached sets of items are equal.

Besides, `PDAState`s will not exist independently. They will be referred to by a `PushDownAutomaton` (See page 651) instance. One of the constraints on such an automaton states that it cannot contain more than a single state with content-wise (as opposed to reference-wise only) equivalent sets of `FlatItem`s (See page 621).

637 ⟨`PDAState` *class definition* 637⟩≡

CLASS `PDAState;`
  INHERITS `HashElement;`

  ⟨*Attaching a* `FlatItemSet` *to a* `PDAState`*, definition* 639b⟩
  ⟨*Displaying a* `PDAState` *on an* `OutputStream`*, definition* 640b⟩

  ⟨*Adding a possible reductions to a state, definition* 647b⟩
  ⟨`SortedByTransition` *method definition* 646b⟩

  ⟨*Attaching transitions, definition* 647d⟩
  ⟨*Numbering* `PDAState` *instances, definition* 648c⟩

END `PDAState;`

$LRAction$, as defined below, synthetizes the actions that can be taken when in a LR state: the automaton can *shift* to another state, represented by $\rho PDAState$, or it can *reduce* a right hand side, represented by $\rho FlatRHS$.

$LRAction = \rho PDAState$ (See the next page)
$\cup \rho FlatRHS$ (See page 511)

PDAState

$Items,$
$FinalItems : \mathbb{P}\,\rho FlatItem$ (See page 623)
$Kernel : \mathbb{B}$

$Items \neq \phi$
$FinalItems = \{x \in Items \bullet \delta x.FarRight = \text{true}\}$

The following constraint expresses the fact that the $Items$ of a $PDAState$ must be closed.

$\forall\, i \in Items \bullet$
$\quad \delta i.Closure \subset Items$
$Kernel = \text{true} \Leftrightarrow \forall\, i \in Items \bullet i.Kernel = \text{true}$

$Transition$ as defined here is just the generalization to $PDAState$ os the $Transition$ function as defined for $FlatItem$s (See page 623).

$Transition : Symbol \mapsto \rho PDAState$

$\forall\, s \in Symbol \bullet$
$\quad \delta Transition(s).Items = \bigcup_{i \in Items} \delta i.Transition(s)$

$Actions$ returns the possible actions (See the page before) that can be taken when in a $PDAState$, given a grammar and a lookahead symbol. These actions can be:

- A *shift*, represented by the $\rho PDAState$ returned by $Transition$ when provided with the symbol
- A *reduction*, represented by those of the $FlatRHS$ for which the present $PDAState$ includes the rightmost item, and for which the lookahead symbol is in the *FOLLOW* set, indicating that a reduction is possible.

$Actions : (\mathbb{P}\, FlatRule \times TokenCode) \mapsto iseq\; LRAction$ (See the page before)

$\forall\, t \in TokenCode, Grammar \in \mathbb{P}\, FlatRule \bullet$
$\quad S \in \text{seq}\,\rho PDAState \bullet$
$\qquad Transitions(t) \neq \Lambda \Rightarrow S = \langle Transitions(t) \rangle$
$\qquad \wedge\; Transitions(t) = \Lambda \Rightarrow S = \langle\,\rangle$
$\quad R \in \text{seq}\,\rho FlatRHS \bullet$
$\qquad \text{ran}\, R = \{x \in Reductions \mid$
$\qquad\qquad t \in (\delta(\delta x).Enclosing).FollowSet(Grammar)\}$
$\quad Actions(Grammar, t) = S \frown R$

$$\begin{array}{l} ReachableStates : \mathbb{P}\,\rho PDAState \\ \hline r \in ReachableStates \Leftrightarrow \\ \quad \exists\, s \in Symbol \bullet \\ \qquad r \in Transition(s) \\ \qquad \vee \\ \qquad \exists\, r' \in Transition(s) \bullet \\ \qquad\quad r \in \delta r'.ReachableStates \end{array}$$

$Reductions$ yields the set of $FlatRHS$ (See page 511) that can be reduced when in the state denoted by $self$. This is simply done by taking the enclosing RHS's of all the final items, that is, the items which lie on the rightmost position of the RHS.

$$\begin{array}{l} Reductions : \mathbb{P}\,\rho FlatRHS \\ \hline Reductions = \bigcup_{f \in FinalItems} \delta f.Enclosing \end{array}$$

639a ⟨`PDAState` *class implementation* 639a⟩≡

```
CLASS PDAState;
  INHERITS HashElement;

  ⟨Attaching a FlatItemSet to a PDAState, implementation 640a⟩
  ⟨Displaying a PDAState on an OutputStream, implementation 640c⟩
  ⟨Generating the code to handle the corresponding PDA state 642a⟩
  ⟨Computing the set of terminals which are valid in this PDA state 646a⟩
  ⟨SortedByTransition method implementation 646c⟩
  ⟨Adding a possible reductions to a state, implementation 647c⟩
  ⟨Attaching transitions, implementation 648a⟩
  ⟨Numbering PDAState instances, implementation 648d⟩
  ⟨Build the corresponding array for interpreted parsers 649c⟩

END PDAState;
```

639b ⟨*Attaching a* `FlatItemSet` *to a* `PDAState`, *definition* 639b⟩≡

```
REDEFINE METHOD CREATE(Set: FlatItemSet);
METHOD Set: FlatItemSet;
```

In order to implement this feature efficiently, `PDAState` inherits from `HashElement`, and its hash value is computed as the embedded `FlatItemSet`'s (See page 630) hash value for value-based retrieval.

640a ⟨*Attaching a* `FlatItemSet` *to a* `PDAState`*, implementation* 640a⟩≡

```
VAR
  TheSet: FlatItemSet;

REDEFINE METHOD HashValue: INTEGER;
  BEGIN
  RESULT ← TheSet.HashValue;
  END HashValue;

REDEFINE METHOD CREATE(Set: FlatItemSet);
  BEGIN
  BASE;
  THIS.TheSet ← Set;
  TheTermTrans.CREATE;
  TheNonTermTrans.CREATE;
  TheReductions.CREATE;
  END CREATE;

METHOD Set: FlatItemSet;
  BEGIN
  RESULT ← TheSet;
  END Set;
```

640b ⟨*Displaying a* `PDAState` *on an* `OutputStream`*, definition* 640b⟩≡

```
METHOD Dump (Output: OutputStream);
```

Actually displaying the set happens my means of the `Dump` method (See page 633).

640c ⟨*Displaying a* `PDAState` *on an* `OutputStream`*, implementation* 640c⟩≡

```
METHOD Dump (Output: OutputStream);
  BEGIN
  ⟨Dump the state number, if relevant 641a⟩
  TheSet.Dump (Output);
  ⟨Dump terminal transitions attached to THIS 641b⟩
  ⟨Dump non-terminal transitions attached to THIS 641c⟩
  ⟨Dump reductions attached to THIS 641d⟩
  END Dump;
```

The state number is produced if the `PDAStates` have been numbered, and `Nr`'s initial value of -1 has been overridden.

641a ⟨*Dump the state number, if relevant* 641a⟩≡

```
IF Nr ≥ 0 THEN
  Output.WriteString (“State: ”);
  Output.WriteInt (Nr, 0);
  Output.WriteLn;
  END;
```

We make use of `TerminalTransition`'s ability to dump themselves (See method *DuraTransitions.TerminalTransition.Dump* on page 661) . . .

641b ⟨*Dump terminal transitions attached to* `THIS` 641b⟩≡

```
IF TheTermTrans ≠ Λ THEN
  Output.WriteLine (“Terminal transitions:”);
  FOR TermTran ∈ TheTermTrans DO
    Output.WriteString (“  ”);
    TermTran.Dump (Output);
    END;
  END;
```

Similarly, for `NonTerminalTransition`, (See method *DuraTransitions.NonTerminalTransition.Dump* on page 662) . . .

641c ⟨*Dump non-terminal transitions attached to* `THIS` 641c⟩≡

```
IF TheNonTermTrans.Size > 0 THEN
  Output.WriteLine (“Non-terminal transitions:”);
  FOR NTermTran ∈ TheNonTermTrans DO
    Output.WriteString (“  ”);
    NTermTran.Dump (Output);
    END;
  END;
```

641d ⟨*Dump reductions attached to* `THIS` 641d⟩≡

```
IF TheReductions.Size > 0 THEN
  Output.WriteLine (“Reductions:”);
  FOR Red ∈ TheReductions DO
    Output.WriteString (“  ”);
    Red.Dump (Output);
    END;
  END;
```

7.16.1.1 Code generation

GenerateState generates the code that handles the behaviour of the automaton when in the state denoted by THIS. It does so by first gathering all the tokens for which a valid transition exists in this state (See method *DuraPDA.PDAState.ValidTerminals* on page 646).

642a ⟨*Generating the code to handle the corresponding PDA state* 642a⟩≡

```
METHOD GenerateState (Output: DuraOutputStream);
  VAR
    NextAction, Code: INTEGER;
    Toks: ARRAY OF INTEGER;
    TermTran : TerminalTransition;
  BEGIN
  TheTermTrans.Sort;
  TheReductions.Sort;
  Toks ← ValidTerminals.TokensRow;
  ASSERT ∀x ∈ Toks • x ≥ 0;
  IF Toks=Λ THEN
    ⟨Degenerated case, simply fail 642b⟩
   ELSE
    ASSERT Toks.SIZE > 0;
    ⟨There is at least a valid token in this state 642c⟩
    END;
  END GenerateState;
```

If no such valid token exists, the parser must fail when in this state by setting an ad hoc flag to FALSE (See *Parser skeleton.Parser skeleton.TheFailure* on page 869).

642b ⟨*Degenerated case, simply fail* 642b⟩≡

```
Output.WriteLine (“TheFailure := TRUE;”);
```

Otherwise, there is at least a single token for which an outgoing transition exists for this state (See method *YLexicalAnalyzer.YLexicalAnalyzer.CurrentToken* on page 187).

642c ⟨*There is at least a valid token in this state* 642c⟩≡

```
Output.WriteLine (“CASE TheLex.CurrentToken OF”);
Output.IncIndent(2);
FOR j ∈ 0 .. Toks.SIZE - 1 | Toks_j ≥ 0 DO
  Code ← Toks_j;
  ⟨Consider transitions on Code 643a⟩
  END;
Output.Else;
Output.WriteLine (“TheFailure := TRUE;”);
Output.End;
Output.DecIndent(2);
```

643a ⟨*Consider transitions on* `Code` 643a⟩≡
```
TermTran ← GetTerminalTransition (Code);
Output.WriteInt (Code, 0);
⟨Consider possible other tokens with the same transitions 643b⟩
Output.WriteLine (“:”);
Output.IncIndent (2);
Output.WriteLine (“CASE TheActionNr OF”);
Output.IncIndent (2);
```

When looking for another token which would generate the same transitions, I only have to scan forward, since the mergeability property is an equivalence relationship, and `Code` would have been merged with any equivalent preceeding token when that preceeding token has been analyzed itself. Checking whether two transitions on a state can be merged is performed by the `Mergeable` method (See page 645).

643b ⟨*Consider possible other tokens with the same transitions* 643b⟩≡
```
FOR i ∈ j + 1 .. Toks.SIZE - 1 DO
  IF (Toks_i ≥ 0) ∧ Mergeable(Code, Toks_i) THEN
    Output.WriteChar (“,”);
    Output.WriteInt (Toks_i, 0);
    Toks_i ← -1;
    END;
  END;
```

We first consider the case where a shift operation can be performed when encountering `Code` in this state . . .

643c ⟨*Consider transitions on* `Code` 643a⟩+≡
```
IF TermTran ≠ Λ THEN
  ⟨There is a shift possible, handle it first 643d⟩
  NextAction ← 1;
 ELSE
  NextAction ← 0;
  END;
```

We generate a call to the `Shift` method defined in the skeleton (See page 875) providing it with the state number it must shift to, extracted from the (by construction, unique) terminal transition that goes from this state.

643d ⟨*There is a shift possible, handle it first* 643d⟩≡
```
Output.WriteLine (“0:”);
Output.IncIndent (2);
Output.WriteString (“Shift(”);
Output.WriteInt (TermTran.Two.Nr, 0);
Output.WriteLine (“);”);
Output.End;
```

We now consider the possible reductions. We extract the right hand sides that can be reduced in front of a `Code` token, and for each of these, produce a call to the corresponding reduce method (See method *DuraFlat.FlatRHS.GenerateReduceMethodCall* on page 526). `NextAction` is a local variable that starts at 0, and which is incremented for each different possible action in this state.

644a ⟨*Consider transitions on* `Code` 643a⟩+≡

```
FOR RHS ∈ TheReductions |
  ⟨RHS can be reduced on token Code 644c⟩
        DO
  Output.WriteInt (NextAction, 0);
  Output.WriteLine ("':");
  Output.IncIndent (2);
  RHS.GenerateReduceMethodCall(Output);
  Output.End;
  NextAction ← NextAction + 1;
  END;
```

We've been through all the actions that could be taken when encountering a token `Code` in this state. We set `TheFailure` (See page 869) to `FALSE` to signal that the parser must start backtracking.

644b ⟨*Consider transitions on* `Code` 643a⟩+≡

```
Output.Else;
Output.WriteLine ("TheFailure := TRUE;");
Output.End;
Output.End;
```

Basically, a right hand side can be reduced on a token if its attached rule's *FOLLOW* set includes that token. In other words, we accept a reduction if there is at least a possibility that something can be done with the current token after that the reduction has been performed.

644c ⟨`RHS` *can be reduced on token* `Code` 644c⟩≡

```
(RHS.GetFlatRule.FollowSet ≠ Λ) ∧
(RHS.GetFlatRule.FollowSet_Code
```

A PDA state has two mergeable transitions on two codes when:

- If one of the two codes is attached to a shift transition, then, both codes must have shift transitions, to the same target state.
- All possible reduces on one of the transitions must be applicable on the other one as well.

645 ⟨*Generating the code to handle the corresponding PDA state* 642a⟩+≡

```
METHOD Mergeable (CodeOne, CodeTwo: INTEGER): BOOLEAN;
  VAR
    TheSet: TerminalSet;
    TermTran1, TermTran2: TerminalTransition;
  BEGIN
  ASSERT CodeOne ≥ 0;
  ASSERT CodeTwo ≥ 0;
  TermTran1 ← GetTerminalTransition (CodeOne);
  TermTran2 ← GetTerminalTransition (CodeTwo);
  RESULT ← ((TermTran1 = Λ) ⇔ (TermTran2 = Λ))
            ∧
             ((TermTran1 ≠ Λ) ⇒ TermTran1.Two = TermTran2.Two);
  FOR Red ∈ TheReductions WHILE RESULT DO
    TheSet ← Red.GetFlatRule.FollowSet;
    RESULT ← TheSet_CodeOne ⇔ TheSet_CodeTwo;
    END;
  END Mergeable;
```

The set of valid terminals in a PDA state is made of two classes:

- The terminals which are attached to a shift transition from this state
- The terminals that can follow a non-terminal that can be reduced in the PDA state. Such terminals are computed in the *FOLLOW* set attached to the non-terminal.

The method in this chunk uses lazy evaluation. On first invocation, the computed set of terminals is stored in `TheValidTerminals` so that further invocation of this method can return this set without further computation.

646a ⟨*Computing the set of terminals which are valid in this PDA state* 646a⟩≡
```
VAR
  TheValidTerminals: TerminalSet;

METHOD ValidTerminals: TerminalSet;
  BEGIN
  IF TheValidTerminals = Λ THEN
    TheValidTerminals.CREATE;
    FOR Red ∈ TheReductions DO
      TheValidTerminals.Or (Red.GetFlatRule.FollowSet);
      END;
    FOR Tran ∈ TheTermTrans DO
      TheValidTerminals.Set (Tran.LexCode);
      END;
    END;
  RESULT ← TheValidTerminals;
  END ValidTerminals;
```

646b ⟨`SortedByTransition` *method definition* 646b⟩≡
```
METHOD SortedByTransition: ArrayList(FlatItem);
```

646c ⟨`SortedByTransition` *method implementation* 646c⟩≡
```
METHOD SortedByTransition: ArrayList(FlatItem);
  BEGIN
  RESULT ← TheSet.BuildList;
  ⟨Sort the list 647a⟩
  END SortedByTransition;
```

At this stage, sorting the list of items might sound like back art. In fact, it is pretty simple. The comparison method defined as part of the `FlatItemList` class (See method *DuraItem.FlatItemList.ShouldSwap* on page 629) works in stages:

- first, put final items, that is, items that are at the rightmost position within their grammar rule
- then, items for which the following token to parse is a non-terminal
- Finally, the items for which the following token is a terminal.

Items related to tokens (resp. non-terminals) are sorted in such a way that all the transitions on a given token (resp. non-terminal) will be grouped in successive indexes within the list.

647a ⟨*Sort the list* 647a⟩≡

```
RESULT.Sort;
```

7.16.1.2 Attaching transitions

It should be clear now that a `PDAState` is going to represent a state in push down automaton (PDA). Several actions can be taken in each of these states, for instance, reducing to a rule, or more specifically, to the right hand side attached to a rule.

647b ⟨*Adding a possible reductions to a state, definition* 647b⟩≡

```
METHOD AppendReduction (RHS: FlatRHS);
```

647c ⟨*Adding a possible reductions to a state, implementation* 647c⟩≡

```
VAR
  TheReductions: RHSList;

METHOD AppendReduction (RHS: FlatRHS);
  BEGIN
  TheReductions.Append (RHS);
  END AppendReduction;
```

647d ⟨*Attaching transitions, definition* 647d⟩≡

```
METHOD AppendTerminalTransition (TermTran: TerminalTransition);
METHOD AppendNonTerminalTransition (NonTermTran: NonTerminalTransition);
```

648a ⟨*Attaching transitions, implementation* 648a⟩≡

```
VAR
  TheTermTrans: TerminalTransitionList;
  TheNonTermTrans: NonTerminalTransitionList;

METHOD AppendTerminalTransition (TermTran: TerminalTransition);
  BEGIN
  TheTermTrans.Append (TermTran);
  END AppendTerminalTransition;

METHOD AppendNonTerminalTransition (NonTermTran: NonTerminalTransition);
  BEGIN
  TheNonTermTrans.Append (NonTermTran);
  END AppendNonTerminalTransition;

METHOD NonTerminalTransitions: NonTerminalTransitionList;
  BEGIN
  RESULT ← TheNonTermTrans;
  END NonTerminalTransitions;
```

YAFL

This chunk uses a YAFL quantifier that returns the first element in a list that matches a given condition. We stop as soon as there is one, since there is at most a single `TerminalTransition` for a given code in any given state.

648b ⟨*Attaching transitions, implementation* 648a⟩+≡

```
METHOD GetTerminalTransition (LexCode: INTEGER): TerminalTransition;
  BEGIN
  RESULT ← FIRST t ∈ TheTermTrans • t.LexCode = LexCode;
  END GetTerminalTransition;
```

648c ⟨*Numbering* `PDAState` *instances, definition* 648c⟩≡

```
METHOD Nr: INTEGER;
```

648d ⟨*Numbering* `PDAState` *instances, implementation* 648d⟩≡

```
VAR
  TheNr: INTEGER;

METHOD SetNr (Nr: INTEGER);
  BEGIN
  TheNr ← Nr;
  END SetNr;

METHOD Nr: INTEGER;
  BEGIN
  RESULT ← TheNr;
  END Nr;
```

`StoreArray` builds a representation of the various actions that can be taken in a state as an array of integers, and stores them in an `IntDictionary` (See page 764). The resulting index is then returned so that the caller (See method *DuraPDA.PushDownAutomaton.StoreArray* on page 656) can gather them for all its attached states in yet another integer array which will be stored in the very same `IntDictionary`.

The array which is stored is indexed by token code. If the n^{th} entry is 0, it indicates that there is no valid action when encountering token n in this state. If the n^{th} entry is non zero, it is the index returned by the `IntDictionary` when storing an array of integers that represent the actions that can be taken when encountering token n in this state.

The array of these actions is structured as follows:

- If a terminal transition (understand: a shift) is possible, the negated target state is set as the first value in the array

649a ⟨*Store the negated target state for shift* 649a⟩≡

```
Temp₀ ← -TermTran.Two.Nr;
```

- The rest of the array is filled with the (positive) number of the various right-hand sides that can be reduced when in this state and encountering a n token.

649b ⟨*Store all the possible reduction that can occur* 649b⟩≡

```
FOR i ∈ 0 .. RHSCount - 1 DO
  Temp[Offset + i] ← RHSs_i.Nr;
  END;
```

649c ⟨*Build the corresponding array for interpreted parsers* 649c⟩≡

```
METHOD StoreArray(Dict: IntDictionary): INTEGER;
  VAR
    Offset: INTEGER;
    Temp, Res, Toks: ARRAY OF INTEGER;
    TermTran : TerminalTransition;
    RHSs: ARRAY OF FlatRHS;
    RHSCount: INTEGER;
  BEGIN
  TheTermTrans.Sort;
  TheReductions.Sort;
  Toks ← ValidTerminals.TokensRow;
  ASSERT ∀x ∈ Toks • x ≥ 0;
  IF Toks ≠ Λ THEN
    Res.CREATE (DuraCfg.LastToken + 1);
    FOR Code ∈ Toks DO
      ⟨Build array for Code in Temp 650a⟩
      Res_Code ← Dict.StoreArray (Temp);
      END;
    RESULT ← Dict.StoreArray(Res);
    END;
  END StoreArray;
```

Consider all the reductions . . .

650a ⟨*Build array for* `Code` *in* `Temp` 650a⟩≡

```
RHSs ← ALL RHS ∈ TheReductions •
    ⟨RHS can be reduced on token Code 644c⟩;
IF RHSs ≠ Λ THEN
  RHSCount ← RHSs.SIZE;
 ELSE
  RHSCount ← 0;
  END;
```

And the (at most !) single terminal transition . . .

650b ⟨*Build array for* `Code` *in* `Temp` 650a⟩+≡

```
TermTran ← GetTerminalTransition (Code);
IF TermTran ≠ Λ THEN
  Offset ← 1;
  Temp.CREATE (1 + RHSCount);
  ⟨Store the negated target state for shift 649a⟩
 ELSE
  Offset ← 0;
  Temp.CREATE (RHSCount);
  END;
⟨Store all the possible reduction that can occur 649b⟩
```

7.16.2 `PDAStateList`

`PDAStateList` inherits from `ArrayList` and redefines the `ShouldSwap` method so that lists of `PDAState`'s can be sorted by state number.

650c ⟨`PDAStateList` *class definition* 650c⟩≡

```
CLASS PDAStateList;
  INHERITS ArrayList(PDAState);

END PDAStateList;
```

650d ⟨`PDAStateList` *class implementation* 650d⟩≡

```
CLASS PDAStateList;
  INHERITS ArrayList(PDAState);

  REDEFINE METHOD ShouldSwap (One, Two: PDAState): BOOLEAN;
    BEGIN
    RESULT ← One.Nr > Two.Nr;
    END ShouldSwap;

END PDAStateList;
```

7.16.3 `PushDownAutomaton`

`PushDownAutomaton` implements a dictionary of item sets. In other words, its major purpose is to keep track of which of such `PDAState`s have been stored in order to avoid storing them twice.

651 ⟨`PushDownAutomaton` *class definition* 651⟩≡

```
CLASS PushDownAutomaton;
  REDEFINE METHOD CREATE;

  METHOD Exists (FSet: FlatItemSet): PDAState;
  METHOD Add(State: PDAState);
  METHOD Renumber(FinalItem: FlatItem);
  METHOD Dump(Output: OutputStream);
  METHOD DumpStatistics;
  METHOD Size: INTEGER;

  ⟨Generate the state machine, definition 654c⟩
  ⟨Manage the list of all non-terminal transitions, definition 655b⟩
  ⟨Handle the corresponding array for interpreted parsers, definition 656c⟩
  ⟨Get the states that include one or more FlatItem's, definition 655d⟩

END PushDownAutomaton;
```

PushDownAutomaton

Since the *FlatItems* (See page 623) it contains fully determines a *PDAState* (See page 638), an automaton cannot have two states with exactly the same *FlatItem*s.

$$States : \mathbb{P}\, PDAState$$

$$\forall s_1, s_2 \in States \mid s_1 \neq s_2 \bullet s_1.Items \neq s_2.Items$$

The following two functions are ancillary utilities to return the set of *PDAState*s which contain a given *FlatItem*, or which have a non-empty intersection with a set of *FlatItem*s.

$$StatesByItem : FlatItem \mapsto \mathbb{P}\, PDAState$$
$$StatesByItemSet : \mathbb{P}\, FlatItem \mapsto \mathbb{P}\, PDAState$$

$$\forall i \in FlatItem \bullet$$
$$\quad StatesByItem(i) = \{s \in States \mid i \in s.Items\}$$
$$\forall s \in \mathbb{P}\, FlatItem \bullet$$
$$\quad StatesByItemSet(s) = \bigcup_{i \in s} StatesByItem(i)$$

A `PushDownAutomaton` maintains a dual data structure for the `PDAState`s it contains:

- A sequential order representation, implemented by `TheList`
- A hashtable, to be able to retrieve a `PDAState` based on its attached `FlatItemSet`. It is initialized with a size of 4096 elements, but it can be reallocated if required (See the next page).

652 ⟨`PushDownAutomaton` *class implementation* 652⟩≡

```
CLASS PushDownAutomaton;

   VAR
     TheList: PDAStateList;
     TheHashTable: HashSpace(PDAState);

   CONST
     InitialSize = 4096;

   REDEFINE METHOD CREATE;
     BEGIN
     BASE;
     TheList.CREATE;
     TheHashTable.CREATE (InitialSize);
     END CREATE;

   METHOD Size: INTEGER;
     BEGIN
     RESULT ← TheList.Size;
     END Size;

   ⟨Existence test method 653a⟩
   ⟨Adding a new PDAState 653b⟩
   ⟨Numbering the PDA states in the dictionary 654a⟩
   ⟨Dumping the PDA states onto an output stream 654b⟩
   ⟨Generate the state machine, implementation 655a⟩
   ⟨Manage the list of all non-terminal transitions, implementation 655c⟩
   ⟨Handle the corresponding array for interpreted parsers, implementation 656d⟩
   ⟨Get the states that include one or more FlatItem's, implementation 656a⟩
   ⟨Statistics handling, implementation 657⟩

END PushDownAutomaton;
```

`Exists` returns the `PDAState` in `THIS` which attached `FlatItemSet` matches `FSet` if present, and `VOID` if no such state exists. It uses the fact that `PDAState`'s hash function is computed by taking its attached `FlatItemSet` hash value (See page 640).

The chunk below is very typical of the usage of the YAFL standard library's hash tables: `Find` is used to find the first entry that matches the hash value `Value`, and `Next` is invoked until the required element is found or until there is no more element with this hash value in the hash table.

653a ⟨*Existence test method* 653a⟩≡

```
METHOD Exists (FSet: FlatItemSet): PDAState;
  VAR
    Value: INTEGER;
  BEGIN
  Value ← FSet.HashValue;
  RESULT ← TheHashTable.Find(Value);
  WHILE (RESULT ≠ Λ) ∧ (¬ RESULT.Set.IsEqual(FSet)) DO
    RESULT ← TheHashTable.Next;
    END;
  END Exists;
```

`Add` adds a new `PDAState` to the automaton. It assumes there is no such state yet defined in the automaton.

653b ⟨*Adding a new* `PDAState` 653b⟩≡

```
METHOD Add(State: PDAState);
  VAR
    Flag: BOOLEAN;
  BEGIN
  ASSERT Exists (State.Set) = Λ;
  ⟨Resize the hash table if necessary 653c⟩
  Flag ← TheHashTable.Store(State);
  ASSERT Flag;
  TheList.Append (State);
  END Add;
```

YAFL

`Add` also takes care of the resizing heuristics, based on the number of elements in the hash table compared with its size. If the hash table is filled up to over 50% of its capacity, it is doubled.

653c ⟨*Resize the hash table if necessary* 653c⟩≡

```
IF TheHashTable.Stored * 2 >
           TheHashTable.AllocatedSize THEN
  TheHashTable.Resize (TheHashTable.AllocatedSize * 2);
  END;
```

The following method renumbers the states in the push down automaton. The only non-trivial task it performs is to check whether the final item (which is unique by construction) is included in the state, in which case it received the constant `FinalState` (See page 323).

The method also makes sure that only a single of such states exists.

654a ⟨*Numbering the PDA states in the dictionary* 654a⟩≡

```
METHOD Renumber(FinalItem: FlatItem);
VAR
  Found: BOOLEAN;
  State: PDAState;
BEGIN
FOR i ∈ 0 .. TheList.Size - 1 DO
  State ← TheList_i;
  IF State.Set.IncludesItem (FinalItem) THEN
    ASSERT ¬ Found;
    Found ← TRUE;
    State.SetNr (BaseParser.FinalState);
   ELSE
    State.SetNr (i);
    END;
  END;
END Renumber;
```

Not very interesting . . .

654b ⟨*Dumping the PDA states onto an output stream* 654b⟩≡

```
METHOD Dump (Output: OutputStream);
  BEGIN
  Output.WriteLn;
  FOR El ∈ TheList DO
    Output.WriteLn;
    El.Dump (Output);
    END;
  END Dump;
```

654c ⟨*Generate the state machine, definition* 654c⟩≡

```
METHOD GenerateStateMachine (Output: DuraOutputStream);
```

GenerateStateMachine generates the body of a YAFL CASE which would select the action to execute depending on a state number. It relies on the GenerateState method (See page 642) for the generation of the individual state logics.

655a ⟨*Generate the state machine, implementation* 655a⟩≡

```
METHOD GenerateStateMachine (Output: DuraOutputStream);
  BEGIN
  FOR i ∈ 0 .. TheList.Size - 1 DO
    Output.WriteLine (``-----------------------------'');
    Output.WriteInt (i, 0);
    Output.WriteLine (``:'');
    Output.IncIndent (2);
    TheList_i.GenerateState (Output);
    Output.WriteLine (``END;'');
    Output.DecIndent (2);
    END;
  END GenerateStateMachine;
```

The NonTerminalTransitions method returns a list that includes all the non-terminal transition in the push down automaton. Its value is built on its first invocation, and returned without further processing on all subsequent calls.

Instead of returning a plain List of NonTerminalTransitions, this method returns a NonTerminalTransitionList (See page 663) that supports advanced sorting abilities.

655b ⟨*Manage the list of all non-terminal transitions, definition* 655b⟩≡

```
METHOD NonTerminalTransitions: NonTerminalTransitionList;
```

655c ⟨*Manage the list of all non-terminal transitions, implementation* 655c⟩≡

```
VAR
  TheNonTerminalTransitions: NonTerminalTransitionList;

METHOD NonTerminalTransitions: NonTerminalTransitionList;
  BEGIN
  IF TheNonTerminalTransitions = Λ THEN
    TheNonTerminalTransitions.CREATE;
    FOR El ∈ TheList DO
      FOR El2 ∈ El.NonTerminalTransitions  DO
        TheNonTerminalTransitions.Append (El2);
        END;
      END;
    END;
  RESULT ← TheNonTerminalTransitions;
  END NonTerminalTransitions;
```

655d ⟨*Get the states that include one or more* FlatItem*'s, definition* 655d⟩≡

```
METHOD StatesByItem (f: FlatItem): ARRAY OF PDAState;
METHOD StatesByItemSet (FSet: FlatItemSet):
                                   ARRAY OF PDAState;
```

`StatesByItem` (resp. `StatesByItemSet`) is based on the fact that `PDAState`'s are little more than sets of `FlatItem` (See page 621) and returns all the states in an automaton that include a given item (resp. that have a non-empty intersection with a set of items).

656a ⟨*Get the states that include one or more* `FlatItem`*'s, implementation* 656a⟩≡

```
METHOD StatesByItem (f: FlatItem): ARRAY OF PDAState;
  BEGIN
  RESULT ← ALL El ∈ TheList • El.Set.IncludesItem(f);
  END StatesByItem;
```

656b ⟨*Get the states that include one or more* `FlatItem`*'s, implementation* 656a⟩+≡

```
METHOD StatesByItemSet (FSet: FlatItemSet): ARRAY OF PDAState;
  VAR
    l: ArrayList(PDAState);
  BEGIN
  -- RESULT := ALL El IN TheList :- ((El.Set <> VOID) AND El.Set.Intersects(FSet));
  FOR El ∈ TheList DO
    IF ((El.Set ≠ Λ) ∧ El.Set.Intersects(FSet)) THEN
      IF l = Λ THEN
        l.CREATE;
         END;
      l.Append(El);
      END;
    END;
  IF l ≠ Λ THEN
    RESULT ← l.Row;
    END;
  END StatesByItemSet;
```

656c ⟨*Handle the corresponding array for interpreted parsers, definition* 656c⟩≡

```
METHOD StoreArray(Dict: IntDictionary): INTEGER;
```

`StoreArray` builds an interpretable representation of the automaton, and stores it onto an `IntDictionary` (See page 764). It calls the similary named method for each of its `PDAStates` (See page 649) and then stores the array made of the so gathered indexes (See page 782).

656d ⟨*Handle the corresponding array for interpreted parsers, implementation* 656d⟩≡

```
METHOD StoreArray(Dict: IntDictionary): INTEGER;
  VAR
    Temp: ARRAY OF INTEGER;
  BEGIN
  Temp.CREATE (TheList.Size);
  FOR i ∈ 0 .. Temp.SIZE-1 DO
    Temp_i ← TheList_i;
    END;
  RESULT ← Dict.StoreArray(Temp);
  END StoreArray;
```

657 ⟨*Statistics handling, implementation* 657⟩≡

```
METHOD DumpStatistics;
  VAR
    x, Avg: REAL;
    Count, j, Min, Max, Sum, Sum2: INTEGER;
  BEGIN
  Min ← 9999999;
  Max ← -9999999;
  Count ← TheList.Size;
  FOR El ∈ TheList DO
    j ← El.Set.UsedBits;
    IF j < Min THEN
      Min ← j;
      END;
    IF j > Max THEN
      Max ← j;
      END;
    Sum ← Sum + j;
    Sum2 ← Sum2 + (j*j);
    END;
  StdOut.WriteString (“Min: ”);
  StdOut.WriteInt (Min, 0);
  StdOut.WriteString (“ Max: ”);
  StdOut.WriteInt (Max, 0);
  StdOut.WriteString (“ Avg: ”);
  Avg ← SYSTEM.FLOAT(Sum)/SYSTEM.FLOAT(Count);
  StdOut.WriteReal (Avg, 8, 2);
  StdOut.WriteString (“ StdDev: ”);
  x ← SYSTEM.FLOAT(Sum2)/SYSTEM.FLOAT(Count) - (Avg * Avg);
  ASSERT x ≥ 0.0;
  StdOut.WriteReal (Math.Sqrt(x), 8, 2);
  StdOut.WriteLn;
  END DumpStatistics;
```

658a ⟨DuraPDA *definition import list* 658a⟩≡

```
IMPORT ArrayList;
FROM DuraFlat IMPORT FlatRHS, FlatRuleComponent;
FROM DuraTransitions IMPORT TerminalTransition,
                         NonTerminalTransition,
                            NonTerminalTransitionList,
                            TerminalTransitionList;
FROM DuraIntDictionary IMPORT IntDictionary;
FROM DuraItem IMPORT FlatItem, FlatItemSet, FlatItemList;
FROM DuraOutput IMPORT DuraOutputStream;
FROM Streams IMPORT StdOut, OutputStream;
IMPORT HashElement;
```

658b ⟨DuraPDA *implementation import list* 658b⟩≡

```
FROM DuraBase IMPORT BaseParser;
FROM DuraBitMap IMPORT DuraBitMap;
FROM DuraCfg IMPORT DuraCfg;
FROM DuraIntermediateFormat IMPORT InterRule;
FROM DuraFlat IMPORT FlatTerminal, FlatNonTerminal, TerminalSet, RHSList;
FROM DuraReductions IMPORT FetchNonTerminalReduction;

IMPORT HashSpace;
IMPORT Math;
IMPORT Set;
IMPORT Stack;
IMPORT SYSTEM;
```

7.17 DuraTransitions

This module implements transitions, that is, objects which connect states (See class *DuraPDA.PDAState* on page 636) of our LR automaton, together with some information about the symbol that must be encountered to cause the transition to be fired.

These transitions can be of two kinds, depending on whether the symbol on which they can be fired is a terminal or a non-terminal.

Figure 7.9 Inheritance tree for transitions

```
                              ┌─ TerminalTransition (661)
Transition (660) ─────────────┤
                              └─ NonTerminalTransition (662)
```

This module also implements lists of each of these classes of transitions, essentially to support specific sort methods (by start state, target state, etc.).

⟨*Definition* ⟩≡

```
DEFINITION MODULE DuraTransitions;

IMPORT ArrayList;
IMPORT Couple;
FROM Streams IMPORT OutputStream;

FROM DuraFlat IMPORT FlatRule;
FROM DuraPDA IMPORT PDAState;

  ⟨Transition class definition 660a⟩
  ⟨TerminalTransition class definition 661a⟩
  ⟨NonTerminalTransition class definition 662a⟩

  ⟨TerminalTransitionList class definition 662c⟩
  ⟨NonTerminalTransitionList class definition 663b⟩

END DuraTransitions;
```

⟨*Implementation* ⟩≡
```
IMPLEMENTATION MODULE DuraTransitions;

FROM DuraCfg IMPORT DuraCfg;

  ⟨Transition class implementation 660b⟩
  ⟨TerminalTransition class implementation 661b⟩
  ⟨NonTerminalTransition class implementation 662b⟩

  ⟨TerminalTransitionList class implementation 663a⟩
  ⟨NonTerminalTransitionList class implementation 664a⟩

END DuraTransitions;
```

`Transition` is the base class for all kinds of transitions supported by LR automata. It connects two `PDAStates` (See page 636) by inheriting from the `Couple` classas provided by the YAFL standard library.

YAFL

660a ⟨`Transition` *class definition* 660a⟩≡
```
CLASS Transition;
  INHERITS Couple (PDAState);

  METHOD Dump (Output: OutputStream);
END Transition;
```

660b ⟨`Transition` *class implementation* 660b⟩≡
```
CLASS Transition;
  INHERITS Couple (PDAState);

  METHOD Dump (Output: OutputStream);
    BEGIN
    Output.WriteString (‘‘From state ”);
    Output.WriteInt (One.Nr, 0);
    Output.WriteString (‘‘ to ”);
    Output.WriteInt (Two.Nr, 0);
    Output.WriteString (‘‘ ”);
    END Dump;

END Transition;
```

`TerminalTransition` is a terminal transition on a token which code must match the `LexCode` parameter provided at creation time.

661a ⟨`TerminalTransition` *class definition* 661a⟩≡

```
CLASS TerminalTransition;
  INHERITS Transition;
  REDEFINE METHOD CREATE (From, To: PDAState;
                          LexCode: INTEGER);
  METHOD LexCode: INTEGER;
END TerminalTransition;
```

Well, for the rest, not much to say.

661b ⟨`TerminalTransition` *class implementation* 661b⟩≡

```
CLASS TerminalTransition;
  INHERITS Transition;

  VAR
    TheLexCode: INTEGER;

  REDEFINE METHOD CREATE (From, To: PDAState;
                          LexCode: INTEGER);
    BEGIN
    BASE (From, To);
    TheLexCode ← LexCode;
    END CREATE;

  REDEFINE METHOD Dump (Output: OutputStream);
    BEGIN
    BASE (Output);
    Output.WriteLine (‘‘on lexeme ” +
                      DuraCfg.LexemeImage (TheLexCode));
    END Dump;

  METHOD LexCode: INTEGER;
    BEGIN
    RESULT ← TheLexCode;
    END LexCode;

END TerminalTransition;
```

Similarly, `NonTerminalTransition` is a terminal transition on a non-terminal as defined by a reference to a `FlatRule` (See page 480) provided at creation time.

662a ⟨`NonTerminalTransition` *class definition* 662a⟩≡
```
CLASS NonTerminalTransition;
  INHERITS Transition;
  REDEFINE METHOD CREATE (From, To: PDAState;
                          Rule: FlatRule);
  METHOD Rule: FlatRule;
END NonTerminalTransition;
```

662b ⟨`NonTerminalTransition` *class implementation* 662b⟩≡
```
CLASS NonTerminalTransition;
  INHERITS Transition;

  VAR
    TheRule: FlatRule;

  REDEFINE METHOD CREATE (From, To: PDAState;
                          Rule: FlatRule);
    BEGIN
    ASSERT Rule ≠ Λ;
    BASE (From, To);
    TheRule ← Rule;
    END CREATE;

  METHOD Rule: FlatRule;
    BEGIN
    RESULT ← TheRule;
    END Rule;

  REDEFINE METHOD Dump (Output: OutputStream);
    BEGIN
    BASE (Output);
    Output.WriteString (“on nonterminal ”);
    Output.WriteInt (TheRule.Nr, 0);
    Output.WriteLn;
    END Dump;

END NonTerminalTransition;
```

662c ⟨`TerminalTransitionList` *class definition* 662c⟩≡
```
CLASS TerminalTransitionList;
  INHERITS ArrayList(TerminalTransition);
END TerminalTransitionList;
```

Lists of `TerminalTransition` are to be sorted by token code values in increasing order.

663a ⟨`TerminalTransitionList` *class implementation* 663a⟩≡

```
CLASS TerminalTransitionList;
  INHERITS ArrayList(TerminalTransition);

  REDEFINE METHOD ShouldSwap (One, Two: TerminalTransition): BOOLEAN;
    BEGIN
    RESULT ← One.LexCode > Two.LexCode;
    END ShouldSwap;

END TerminalTransitionList;
```

The `NonTerminalTransitionList` is a rather naive derivation of the `ArrayList` class for handling lists of `NonTerminalTransition` instances. Its only supplementary service is a flexible sort method, where one can indicate the sort mode by means of the `SetSortMethod` prior to calling the `Sort` method. YAFL

663b ⟨`NonTerminalTransitionList` *class definition* 663b⟩≡

```
CLASS NonTerminalTransitionList;
  INHERITS ArrayList(NonTerminalTransition);

  CONST
    BySourceSortMethod = 0;
    ByTargetSortMethod = 1;
    ByTransitionSortMethod = 2;

  METHOD SetSortMethod (SortMethod: INTEGER);
    PRE
      (SortMethod = BySourceSortMethod) ∨
      (SortMethod = ByTargetSortMethod) ∨
      (SortMethod = ByTransitionSortMethod);

END NonTerminalTransitionList;
```

664a ⟨NonTerminalTransitionList *class implementation* 664a⟩≡
```
CLASS NonTerminalTransitionList;
  INHERITS ArrayList(NonTerminalTransition);

  VAR
    TheSortMethod: INTEGER;

  METHOD SetSortMethod (SortMethod: INTEGER);
    BEGIN
    TheSortMethod ← SortMethod;
    END SetSortMethod;

  ⟨ShouldSwap, as redefined by NonTerminalTransitionList 664b⟩

END NonTerminalTransitionList;
```

ShouldSwap compares NonTerminalTransitions according to the value of TheSortMethod which indicates how the list must be sorted.

664b ⟨ShouldSwap, *as redefined by* NonTerminalTransitionList 664b⟩≡
```
REDEFINE METHOD ShouldSwap (One, Two: NonTerminalTransition): BOOLEAN;
  BEGIN
  CASE TheSortMethod OF
    BySourceSortMethod:
      RESULT ← One.One.Nr > Two.One.Nr;
      END;
    ByTargetSortMethod:
      RESULT ← One.Two.Nr > Two.Two.Nr;
      END;
    ByTransitionSortMethod:
      RESULT ← One.Rule.Nr > Two.Rule.Nr;
      END;
    END;
  END ShouldSwap;
```

7.18 DuraGrClause

This module includes the high-level components which implement grammatical extensions in DURA. There is at most a single `GrammarClause` attached to every class (See page 434). Such a `GrammarClause` can include one or more `GrammarRule`s, connected by an implicit disjunction.

Both classes are extensions of the compiler in the sense that they must be integrated in the plain YAFL parse tree. Consequently, they inherit from `BaseNonTerminal` as defined by the YAFL compiler, and redefine a number of methods in order to be able to interact gracefully with the rest of the YAFL compiler.

⟨*Definition* ⟩≡

```
DEFINITION MODULE DuraGrClause;

FROM YaflParser IMPORT NonTerminal;
FROM YaflGCode IMPORT DummyNTCodeGenerator;
FROM YaflNTList IMPORT NTList;
FROM DuraCG IMPORT RuleCodeGenerator;
FROM DuraClDefinition IMPORT DuraClassDefinition;

FROM DuraExpressions IMPORT GrammarExpression;

  ⟨GrammarClause class definition 666a⟩
  ⟨GrammarRule class definition 668a⟩

END DuraGrClause;
```

⟨*Implementation* ⟩≡

```
IMPLEMENTATION MODULE DuraGrClause;

FROM YaflCfg IMPORT YaflCfg;
FROM YaflType IMPORT TypedNonTerminal;
FROM YaflLex IMPORT LexicalAnalyzer;
FROM YaflLiteral IMPORT Literal, IntegerLiteral;

FROM DuraCfg IMPORT DuraCfg;
IMPORT DuraLex;
FROM LookAhead IMPORT LookAhead;
FROM DuraLookAhead IMPORT DuraStreamLookAhead;

  ⟨GrammarClause class implementation 666b⟩
  ⟨GrammarRule class implementation 668b⟩

END DuraGrClause;
```

666a ⟨GrammarClause *class definition* 666a⟩≡

```
CLASS GrammarClause;
  INHERITS NonTerminal(DummyNTCodeGenerator);

  REDEFINE METHOD CREATE (LineNr, ColNr: INTEGER);
  METHOD Rules: NTList(GrammarRule);
END GrammarClause;
```

In terms of its basic services, a GrammarClause is little more than a list of rules, implemented by the Rules method, and the TheRules attribute.

666b ⟨GrammarClause *class implementation* 666b⟩≡

```
CLASS GrammarClause;
  INHERITS NonTerminal(DummyNTCodeGenerator);

  VAR
    TheRules: NTList(GrammarRule);

  REDEFINE METHOD CREATE (LineNr, ColNr: INTEGER);
    BEGIN
    BASE (LineNr, ColNr);
    TheRules.CREATE;
    END CREATE;

  METHOD Rules: NTList(GrammarRule);
    BEGIN
    RESULT ← TheRules;
    END Rules;

  ⟨Integrating GrammarClause in the YAFL compiler 666c⟩

END GrammarClause;
```

YAFL

666c ⟨*Integrating* GrammarClause *in the* YAFL *compiler* 666c⟩≡

```
REDEFINE METHOD SubTree: ARRAY OF NonTerminal;
  BEGIN
  RESULT ← TheRules.SubTree;
  END SubTree;

REDEFINE METHOD WhatAmI: ARRAY OF CHAR;
  BEGIN
  RESULT ← "GrammarClause";
  END WhatAmI;
```

`Parse` parses a `GrammarClause`. If must only be invoked after recognizing the `GRAMMAR` keyword. `Parse` constrains `Lkh` into a `DuraStreamLookAhead` (See page 809) because it will need its `DURA`-specialized services (See method *DuraLookAhead.DuraStreamLookAhead.AcceptGrammarRule* on page 810). Then, until the lookahead is a reserved word other than `CASE`, `AcceptGrammarRule` (See page 810) is called to gather the various grammar rules that must be stored in the `GrammarClause`. We'll stop recognizing the grammar rules when encountering a `METHOD`, `VAR`, `REDEFINE`, `CONST` or `DEFERRED` keyword.

667a ⟨*Integrating* `GrammarClause` *in the* $\mathrm{Y^AF_L}$ *compiler* 666c⟩+≡

```
REDEFINE METHOD Parse (Lkh: LookAhead);
  BEGIN
  ASSERT (Lkh.CurrentToken = LexicalAnalyzer.UdfToken) ∧
         (Lkh.CurrentUdfCode = DuraLex.Grammar);
  Lkh.GetToken;
  WHAT Lkh OF
    IN DuraStreamLookAhead:
      WHILE TAG.Ok ∧
           ((TAG.CurrentToken > TAG.LastReserved) ∨
            (TAG.CurrentToken < TAG.FirstReserved) ∨
            (TAG.CurrentToken = TAG.Case)) DO
        TheRules.Append (TAG.AcceptGrammarRule);
        TAG.Skip (TAG.SemiColon);
        END;
      END;
    END;
  END Parse;
```

$\mathrm{Y^AF_L}$

The following two methods implement the two passes of the $\mathrm{Y^AF_L}$ semantic analysis [31]. They first check whether grammatical information must be analyzed (See method *DuraCfg.DuraCfg.ConsiderGrammarInfo* on page 382), in which case they invoke the corresponding semantic analysis pass on the enclosed grammar rules.

667b ⟨*Integrating* `GrammarClause` *in the* $\mathrm{Y^AF_L}$ *compiler* 666c⟩+≡

```
REDEFINE METHOD Tag;
  BEGIN
  IF DuraCfg.ConsiderGrammarInfo THEN
    TheRules.UniqueTag;
    END;
  END Tag;

REDEFINE METHOD CheckType;
  BEGIN
  IF DuraCfg.ConsiderGrammarInfo THEN
    TheRules.UniqueCheckType;
    END;
  END CheckType;
```

A `GrammarRule` is a grammatical expression (See page 446) together with various attributes: cut level, recovery level, precedence and associativity.

668a ⟨`GrammarRule` *class definition* 668a⟩≡
```
CLASS GrammarRule;
  INHERITS NonTerminal(RuleCodeGenerator);

  METHOD Expr: GrammarExpression;

  ⟨GrammarRule constants 669a⟩

  METHOD Precedence: INTEGER;
  METHOD Associativity: INTEGER;
  METHOD CutLevel: INTEGER;
  METHOD RecoveryLevel: INTEGER;

END GrammarRule;
```

668b ⟨`GrammarRule` *class implementation* 668b⟩≡
```
CLASS GrammarRule;
  INHERITS NonTerminal(RuleCodeGenerator);

  VAR
    TheExpr: GrammarExpression;
    TheCutLevel,
    TheRecoveryLevel,
    TheAssociativity,
    ThePrecedence: INTEGER;
    ThePrecedenceExpression: TypedNonTerminal;

  ⟨Consulting attributes trivially 668c⟩
  ⟨Integrating GrammarRule in the YAFL compiler 670c⟩

END GrammarRule;
```

There is no constraint on the value returned by `Precedence`, since it is used for by the user to specify *relative* precedence level among rules. Its actual value is irrelevant.

668c ⟨*Consulting attributes trivially* 668c⟩≡
```
METHOD Precedence: INTEGER;
  BEGIN
  RESULT ← ThePrecedence;
  END Precedence;
```

669a ⟨GrammarRule *constants* 669a⟩≡

```
CONST
  LeftAssoc = -1;
  NoAssoc = 0;
  RightAssoc = 1;
```

In another language than YAFL, `Associativity` would have returned some form of enumerated type. Ever since [32], the lack of enumerated types has been recognized as a weakness in YAFL's design, as they don't seem to integrate gracefully in the OO world.

669b ⟨*Consulting attributes trivially* 668c⟩+≡

```
METHOD Associativity: INTEGER;
  POST
    (RESULT = THIS.LeftAssoc) ∨
    (RESULT = THIS.NoAssoc)   ∨
    (RESULT = THIS.RightAssoc);
  BEGIN
  RESULT ← TheAssociativity;
  END Associativity;
```

669c ⟨GrammarRule *constants* 669a⟩+≡

```
CONST
  FullCut =    DuraClassDefinition.FullCut;
  DefaultCut = DuraClassDefinition.DefaultCut;
  NoCut =      DuraClassDefinition.NoCut;
```

`CutLevel] would also make interesting use of enumerated types. The constants`
`[[FullCut`, `DefaultCut` and `NoCut` have been redefined locally for convenience (See page 415), to avoid numerous verbose qualifications.

669d ⟨*Consulting attributes trivially* 668c⟩+≡

```
METHOD CutLevel: INTEGER;
  POST
    (RESULT = THIS.FullCut) ∨
    (RESULT = THIS.DefaultCut) ∨
    (RESULT = THIS.NoCut);
  BEGIN
  RESULT ← TheCutLevel;
  END CutLevel;
```

669e ⟨GrammarRule *constants* 669a⟩+≡

```
CONST
  AutomaticRecovery = DuraClassDefinition.AutomaticRecovery;
  DefaultRecovery =   DuraClassDefinition.DefaultRecovery;
  NoRecovery =        DuraClassDefinition.NoRecovery;
```

Bis repetita . . .

670a ⟨*Consulting attributes trivially* 668c⟩+≡

```
METHOD RecoveryLevel: INTEGER;
  POST
    (RESULT = THIS.AutomaticRecovery) ∨
    (RESULT = THIS.DefaultRecovery) ∨
    (RESULT = THIS.NoRecovery);
  BEGIN
  RESULT ← TheRecoveryLevel;
  END RecoveryLevel;
```

670b ⟨*Consulting attributes trivially* 668c⟩+≡

```
METHOD Expr: GrammarExpression;
  BEGIN
  RESULT ← TheExpr;
  END Expr;
```

$Y^{A}F_{L}$

670c ⟨*Integrating* `GrammarRule` *in the* $Y^{A}F_{L}$ *compiler* 670c⟩≡

```
REDEFINE METHOD SubTree: ARRAY OF NonTerminal;
  BEGIN
  RESULT.CREATE (2);
  RESULT_0 ← TheExpr;
  RESULT_1 ← ThePrecedenceExpression;
  END SubTree;

REDEFINE METHOD CheckType;
  BEGIN
  TheExpr.UniqueCheckType;
  END CheckType;
```

`Tag` tags the elements of the sub-tree of the `GrammarRule`, namely, the grammatical expression and the expression that denotes the precedence if present.

$Y^{A}F_{L}$

`GetFolded`, as provided by the plain $Y^{A}F_{L}$ compiler, implements constant folding and returns an object that denotes the compile-time evaluable value of an expression, or `VOID` if the expression cannot be evaluated statically.

670d ⟨*Integrating* `GrammarRule` *in the* $Y^{A}F_{L}$ *compiler* 670c⟩+≡

```
REDEFINE METHOD Tag;
  VAR
    Lit: Literal;
  BEGIN
  TheExpr.UniqueTag;
  IF ThePrecedenceExpression ≠ Λ THEN
    ThePrecedenceExpression.UniqueTag;
    Lit ← ThePrecedenceExpression.GetFolded;
    ⟨Make sure Lit is a non-void integer constant 671a⟩
    END;
  END Tag;
```

This chunk first ensures that `ThePrecedenceExpression` can be evaluated statically, then that this static evaluation yields an integer value – there is nothing so far that would have prevented a user from specifiying a string constant for the precedence expression.

671a ⟨*Make sure* `Lit` *is a non-void integer constant* 671a⟩≡

```
IF Lit = Λ THEN
  ThePrecedenceExpression.Error (''The precedence expression " +
                                 ''must be a constant");
 ELSE
  WHAT Lit OF
    IN IntegerLiteral:
      ThePrecedence ← TAG.Value;
      END;
   ELSE
    ThePrecedenceExpression.Error
              (''The precedence expression " +
               ''must be an integer");
    END;
  END;
```

Parsing a `GrammarRule` is performed by parsing its grammar expression (See method *DuraLookAhead.DuraStreamLookAhead.AcceptGrammarExpression* on page 812), then, by recognizing the various options that might have been specified.

671b ⟨*Integrating* `GrammarRule` *in the* $Y\!AF_L$ *compiler* 670c⟩+≡

```
REDEFINE METHOD Parse (Lkh: LookAhead);
  BEGIN
  WHAT Lkh OF
    IN DuraStreamLookAhead:
      TheExpr ← TAG.AcceptGrammarExpression;
      IF Lkh.CurrentToken = Lkh.Colon THEN
        ⟨Accept additional options on the rule 672⟩
        END;
      END;
    END;
  END Parse;
```

boring, but simple.

672 ⟨*Accept additional options on the rule* 672⟩≡

```
Lkh.GetToken;
WHILE (Lkh.CurrentToken = Lkh.UdfToken) ∧ Lkh.Ok DO
  CASE Lkh.CurrentUdfCode OF
    DuraLex.Precedence:
      Lkh.GetToken;
      ThePrecedenceExpression ← Lkh.AcceptPlainExpr;
      END;
    DuraLex.LeftAssoc:
      Lkh.GetToken;
      TheAssociativity ← LeftAssoc;
      END;
    DuraLex.RightAssoc:
      Lkh.GetToken;
      TheAssociativity ← RightAssoc;
      END;
    DuraLex.Snip:
      Lkh.GetToken;
      TheCutLevel ← FullCut;
      END;
    DuraLex.NoCut:
      Lkh.GetToken;
      TheCutLevel ← NoCut;
      END;
    DuraLex.Recover:
      Lkh.GetToken;
      TheRecoveryLevel ← AutomaticRecovery;
      END;
    DuraLex.NoRecover:
      Lkh.GetToken;
      TheRecoveryLevel ← NoRecovery;
      END;
   ELSE
    Lkh.Error (``Unrecognized token'');
    END;
  END;
```

7.19 DuraIdentifiers

This module implements the `DuraIdent` class which is used to represent identifiers in the context of grammatical expressions (See page 445). Typically, in a grammatical expression such as:

```
GRAMMAR
  TheLeft Becomes TheExpr;

VAR
  TheLeft: Identifier;
  TheExpr: Expression;
```

`TheLeft`, `Becomes` and `TheExpr` as they appear in the grammar clause (*not* in the attribute declarations) will be represented by `DuraIdent` instances in the YAFL parse tree. Existing identifiers as defined within the plain YAFL compiler could not be reused here, because they are too narrow or too wide in scope to meet our purpose.

We need the ability of represent a single identifier or two identifiers glued together using the dot notation using a single class, while YAFL provides a class that represents a single identifier, or *designators* which are far more complex and versatile constructs. For instance, `a[b].c[d].e(10,11,12)` is a valid designator. Fitting such flexible constructs in our limited mold would have been harder than basically writing an ad hoc class, since we only need the possibility of qualifying an class name with a module name, as in `Statement.IfStatement`.

⟨*Definition* ⟩≡

```
DEFINITION MODULE DuraIdentifiers;

⟨DuraIdentifiers definition import list 679b⟩

  CLASS DuraIdent;
    INHERITS NonTerminal(IdentCodeGenerator);

    METHOD FirstIdent: Ident;
    METHOD SecondIdent: Ident;

    ⟨Identifying a DuraIdent, definition 678c⟩

  END DuraIdent;

END DuraIdentifiers;
```

⟨*Implementation* ⟩≡
```
IMPLEMENTATION MODULE DuraIdentifiers;

⟨DuraIdentifiers implementation import list 680⟩

  CLASS DuraIdent;
    INHERITS NonTerminal(IdentCodeGenerator);

    VAR
      TheIsThis: BOOLEAN;
      TheFirstIdent, TheSecondIdent: Ident;
      TheLexeme,
      TheConstant: INTEGER;
      TheAttribute: SingleDataItem;
      TheClass: DuraClassDefinition;

    ⟨Redefining methods from the YAFL compiler 674⟩

    ⟨Tagging a DuraIdent 676a⟩
    ⟨Mundane consultation methods 679a⟩

  END DuraIdent;

END DuraIdentifiers;
```

YAFL

`SubTree` must be redefined to allow generic walkthroughs by the YAFL compiler.

674 ⟨*Redefining methods from the YAFL compiler* 674⟩≡
```
REDEFINE METHOD SubTree: ARRAY OF NonTerminal;
  BEGIN
  RESULT.CREATE(2);
  RESULT[0] ← TheFirstIdent;
  RESULT[1] ← TheSecondIdent;
  END SubTree;

REDEFINE METHOD WhatAmI: ARRAY OF CHAR;
  BEGIN
  RESULT ← "DuraIdent";
  END WhatAmI;
```

Parsing a `DuraIdent` is rather straightforward . . .

675a ⟨*Redefining methods from the YAFL compiler* 674⟩+≡

```
REDEFINE METHOD Parse (Lkh: LookAhead);
  BEGIN
  TheFirstIdent ← Lkh.AcceptIdent;
  SetSon (TheFirstIdent);
  IF Lkh.CurrentToken = LexicalAnalyzer.Dot THEN
    Lkh.GetToken;
    TheSecondIdent ← Lkh.AcceptIdent;
    SetSon (TheSecondIdent);
    END;
  END Parse;
```

YAFL

There is no type checking necessary for `DuraIdent`, but we must redefine the original method to prevent the compiler from issuing a runtime warning message about a `CheckType` method that has not been redefined.

675b ⟨*Redefining methods from the YAFL compiler* 674⟩+≡

```
REDEFINE METHOD CheckType;
  BEGIN
  END CheckType;
```

`Tag` is not trivial here, since it must be able to deal with very different cases:

- `THIS` can refer to a non-terminal class, for a non-terminal transition, as `Statement` in:

```
GRAMMAR
  IfKeyword Condition ThenKeyword
    Statement (ElseKeyword Statement)? EndKeyword
```

- `THIS` can refer to an attribute which type is a non-terminal class, for a non-terminal transition which result must be fetched into this attribute, as `TheCond` in:

```
GRAMMAR
  WhileKeyword TheCond DoKeyword Statement EndKeywords;

VAR
  TheCond: Condition;
```

- `THIS` can refer to `THIS` (`THIS` – sorry, I meant, this – is getting quite surrealistic, don't you think so ?), as in:

```
NONTERMINAL CLASS Expression;
  INHERITS BaseNonTerminal;

GRAMMAR
  LParent THIS := Expression RParent;
```

- `THIS` can refer to a constant defined by the lexer, for a terminal transition, as `Plus` in:

```
GRAMMAR
  Expression Plus Expression;
```

676a ⟨*Tagging a* `DuraIdent` 676a⟩≡

```
  REDEFINE METHOD Tag;
    VAR
      TheModule: DefinitionModule;
      ClDecl: ClassDeclaration;
      TheLiteral: Literal;

      ⟨Local method TryClass that checks for valid class references 678b⟩

    BEGIN
    IF TheSecondIdent = Λ THEN
      ⟨If a single identifier was specified, check for various cases 676b⟩
     ELSE
      ⟨If two identifiers were specified, then it must denote a class 678a⟩
      END;
    END Tag;
```

This chunk attempts to identify what the `DuraIdent` refers to. It first uses `FindLexeme` (See page 377) to see whether a lexeme with that name has been defined.

676b ⟨*If a single identifier was specified, check for various cases* 676b⟩≡

```
  TheLexeme ← DuraCfg.FindLexeme (TheFirstIdent.Data);
  IF TheLexeme ≤ 0 THEN
    ⟨Check the YAF_L tag 677a⟩
    END;
```

Otherwise, it tags the first identifier to see what it refers to. Given a plain YAFL `Ident` that has been tagged, `GetRef` yields what it refers to according to the visibility rules defined by the language. If `GetRef` yields a non-`VOID` value, we can use a `WHAT` control structure to check for the various sorts of objects an identifier can refer to.

677a ⟨*Check the YAFL tag* 677a⟩≡

```
TheFirstIdent.UniqueTag;
IF TheFirstIdent.GetRef = Λ THEN
  Error (‘‘Undefined identifier: ” + TheFirstIdent.Data);
 ELSIF TheFirstIdent.Data = PredefItems.This.Id.Data THEN
  TheIsThis ← TRUE;
 ELSE
  WHAT TheFirstIdent.GetRef OF
    IN ConstDeclaration:
      ⟨It must be a lexeme 677b⟩
      END;
    IN SingleDataItem:
      TheAttribute ← TAG;
      END;
    IN ClassDeclaration:
      TryClass (TAG);
      END;
   ELSE
    Error (‘‘Illegal identifier class”);
    END;
  END;
```

If `THIS` refers to constant declaration, we assume it must be an integer constant that denotes a terminal transition. It is however a very unlikely case, where one would use a constant declaration instead of one of the tokens defined in the lexical specification.

677b ⟨*It must be a lexeme* 677b⟩≡

```
TheLiteral ← TAG.GetFolded;
IF TheLiteral = Λ THEN
  TheFirstIdent.Error (‘‘Cannot fold ” + TheFirstIdent.Data);
 ELSE
  WHAT TheLiteral OF
    IN IntegerLiteral:
      TheConstant ← TAG.Value;
      END;
   ELSE
    TheFirstIdent.Error (‘‘Constant ” + TheFirstIdent.Data +
                          ‘‘ should be integer”);
    END;
  END;
```

If the `DuraIdent` is made of two identifiers, the first must refer to a module, and the second must refer to a class within that module.

678a ⟨*If two identifiers were specified, then it must denote a class* 678a⟩≡

```
TheModule ← ModuleTable.LocateDefinitionModule
                                    (TheFirstIdent.Data);
IF TheModule = Λ THEN
  Error (''Module '' + TheFirstIdent.Data + '' not found'');
 ELSE
  ClDecl ← TheModule.GetClass (TheSecondIdent.Data);
  IF ClDecl = Λ THEN
    Error (''Class '' + TheSecondIdent.Data + '' not found'');
   ELSE
    TryClass (ClDecl);
    END;
  END;
```

`TryClass` takes a class declaration, and makes sure it denotes a public class (private classes cannot be referred to from within grammar clauses) in which case it fetches the corresponding class definition into `TheClass`.

By virtue of creators (See method *DuraCreators.DuraObjectCreator.CreateDefinitionModule* on page 405), all class definitions managed from within DURA will be `DuraClassDefinition` (See page 415) instances.

678b ⟨*Local method* `TryClass` *that checks for valid class references* 678b⟩≡

```
METHOD TryClass (Cl: ClassDeclaration);
  BEGIN
  IF Cl.Public THEN
    WHAT Cl.ClosestCanonic OF
      IN DuraClassDefinition:
        TheClass ← TAG;
        END;
      END;
   ELSE
    Error (''Private class reference'');
    END;
  END TryClass;
```

The following (very simple) methods can be used to test what a `DuraIdent` refers to.

678c ⟨*Identifying a* `DuraIdent`*, definition* 678c⟩≡

```
METHOD IsLexeme: INTEGER;
METHOD IsConstant: INTEGER;
METHOD IsAttribute: SingleDataItem;
METHOD IsClass: DuraClassDefinition;
METHOD IsThis: BOOLEAN;
```

679a ⟨*Mundane consultation methods* 679a⟩≡

```
METHOD FirstIdent: Ident;
  BEGIN
  RESULT ← TheFirstIdent;
  END FirstIdent;

METHOD SecondIdent: Ident;
  BEGIN
  RESULT ← TheSecondIdent;
  END SecondIdent;

METHOD IsLexeme: INTEGER;
  BEGIN
  RESULT ← TheLexeme;
  END IsLexeme;

METHOD IsConstant: INTEGER;
  BEGIN
  RESULT ← TheConstant;
  END IsConstant;

METHOD IsAttribute: SingleDataItem;
  BEGIN
  RESULT ← TheAttribute;
  END IsAttribute;

METHOD IsClass: DuraClassDefinition;
  BEGIN
  RESULT ← TheClass;
  END IsClass;

METHOD IsThis: BOOLEAN;
  BEGIN
  RESULT ← TheIsThis;
  END IsThis;
```

679b ⟨DuraIdentifiers *definition import list* 679b⟩≡

```
FROM YaflClasses IMPORT ClassDeclaration;
FROM YaflDeclarations IMPORT SingleDataItem;
FROM YaflParser       IMPORT NonTerminal;
FROM YaflIdentifiers IMPORT Ident;
IMPORT LookAhead;

FROM DuraClDefinition IMPORT DuraClassDefinition;
FROM DuraCG IMPORT IdentCodeGenerator;
```

680 ⟨DuraIdentifiers *implementation import list* 680⟩≡

```
IMPORT DuraCfg;
IMPORT ModuleTable;
FROM YaflLex IMPORT LexicalAnalyzer;
FROM YaflType IMPORT Type;
FROM YaflModules IMPORT CompilationUnit;
FROM YaflDeclarations IMPORT SingleDataItem, ConstDeclaration;
FROM YaflNTList IMPORT DeclList;
FROM YaflLiteral IMPORT Literal, IntegerLiteral, BooleanLiteral;
FROM YaflModules IMPORT DefinitionModule;
FROM YaflPredefined IMPORT PredefItems;
IMPORT Space;
```

7.20 DuraIntermediateFormat

This module implements `DURA`'s intermediate format. This intermediate format is meant to be generated from grammar rules embedded in common YAFL modules, and to be read back afterwards to be assembled into a monolithic parser.

This intermediate format serves different purposes.

First, it serves a highly pragmatic purpose, by allowing the creation of parsers derived from huge amounts of YAFL source code without having to go through the entire source code every time the parser is to be generated. When dealing with very large and complex parsers, such as `COBOL`, for instance, the parser is regenerated very often, as new grammar rules are added or existing grammar rules are updated. When a module is compiled by `DURA`, its grammatical information is produced in a representation which is of course far more compact than the original YAFL representation.

Then, it allows some optimisations to be tested and exercized on this intermediate representation independantly from the tool that generated it in the first place.

Finally, and perhaps more importantly, this intermediate representation provides independance between the various phases. One can think of cases where the grammar is generated by another tool than a patched YAFL compiler. One can also think of cases where a different backend generates an entirely different kind of parser, for instance, a LL(1) parser, based on the same intermediate representation `DURA` produces.

In order to achieve this level of flexibility, this intermediate representation can be read and written to disk very conveniently, by using yet another representation provided by the `DuraLisp` (See page 796) module. Instead of saving or reading the intermediate representation directly to and from actual files, it translates to and from a Lisp-like representation that can be read and written generically.

This is one of the longest modules in DURA. In literate programming form, it is made of almost 100 pages, even if most of this space is filled with uninteresting implementation details. It contains the classes required to describe all the grammatical information used by DURA, listed in figure 7.10 and 7.11.

Figure 7.10 Inheritance tree for DURA's intermediate format

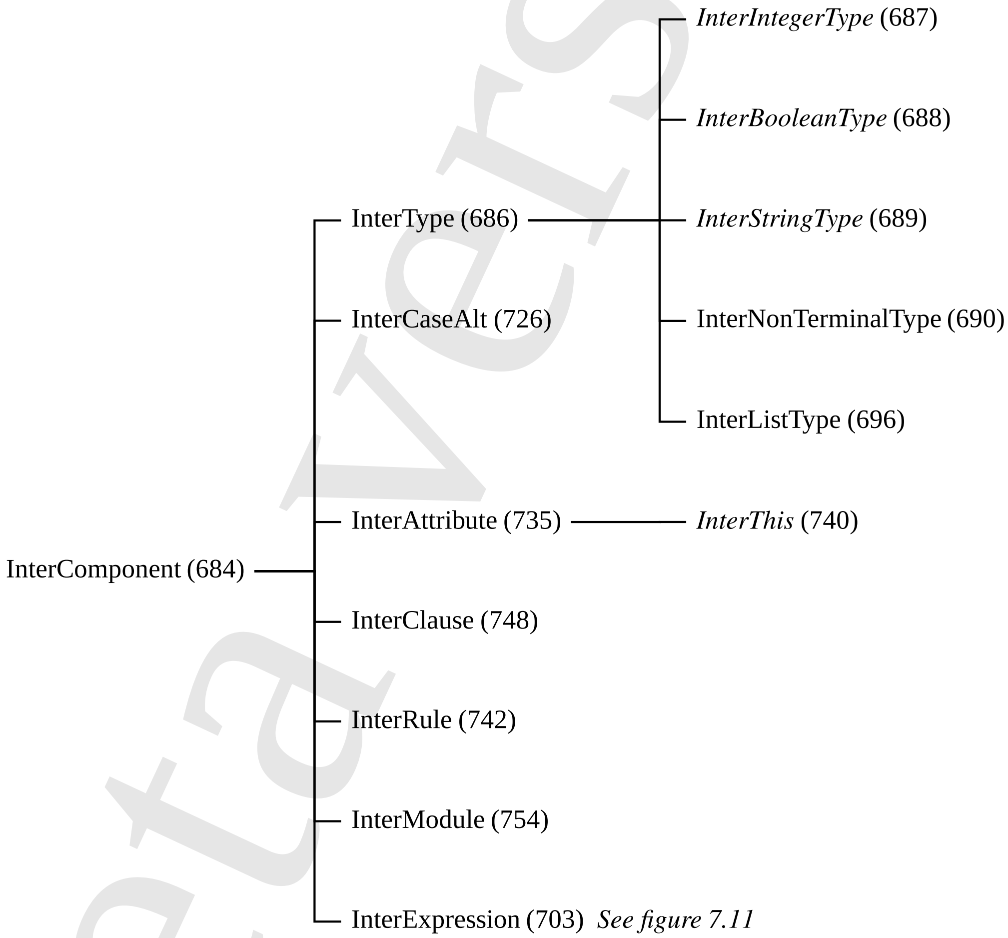
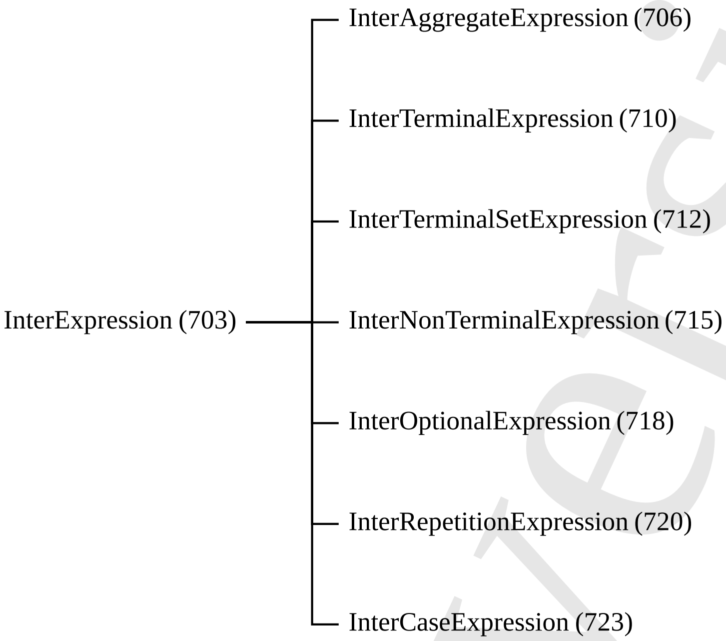


Figure 7.11 Inheritance tree for `DURA`'s intermediate format, continued



⟨*Definition* ⟩≡

DEFINITION MODULE `DuraIntermediateFormat;`

⟨*Definition import list* 762b⟩

⟨`InterComponent` *class definition* 684a⟩
⟨*Intermediate types definition* 686b⟩
⟨*Intermediate expression class definitions* 702b⟩
⟨`InterRule` *class definition* 742⟩
⟨`InterClause` *class definition* 748b⟩
⟨`InterModule` *class definition* 754b⟩

END `DuraIntermediateFormat;`

```
⟨Implementation ⟩≡
  IMPLEMENTATION MODULE DuraIntermediateFormat;

  ⟨Implementation import list 762c⟩

    ⟨InterComponent class implementation 684b⟩
    ⟨InterType base class for types, implementation 687b⟩
    ⟨Predefined types, implementation 688a⟩

    ⟨InterNonTerminalType class implementation 692⟩
    ⟨InterListType class implementation 697a⟩
    ⟨Intermediate expression class implementation 703a⟩
    ⟨Once class that recognizes and parses some InterComponents 757b⟩

    ⟨InterRule class implementation 743a⟩
    ⟨InterClause class implementation 749⟩
    ⟨InterModule class implementation 755a⟩

  END DuraIntermediateFormat;
```

7.20.1 InterComponent

`InterComponent` (for *Intermediate component*) is the base class for all classes used in the intermediate representation.

684a

```
⟨InterComponent class definition 684a⟩≡
  CLASS InterComponent;
    ⟨Generic unparsing 685a⟩

    METHOD GrabSubNodes (l: ArrayList);
  END InterComponent;
```

684b

```
⟨InterComponent class implementation 684b⟩≡
  CLASS InterComponent;

    ⟨Generic unparsing implementation 685b⟩

    DEFERRED METHOD SubTree: ARRAY OF InterComponent;

    ⟨GrabSubNodes method implementation 685d⟩

    END InterComponent;
```

In this context, *unparsing* means taking any item in this intermediate representation, and converting it into an equivalent `Component`, that is, an `Atom` (See page 798) or a `Compound` (See page 801) as defined in the `DuraLisp` module (See page 796).

This intermediate representation can then be saved or loaded from a file very conveniently.

In practice, the actual way to build this unparsed representation will of course depend on the kind of `InterComponent` that is being unparsed. However, the `UnParse` method as described here is not deferred, since it implements lazy evaluation for all `InterComponent`'s unparsing. The first time this method is invoked for a given object, the unparsed representation is computed by `ComputeUnParse` (See the current page), (which is redefined specifically by all `InterComponent`-derived classes) and is saved in the `TheUnParse` attribute, that will be returned for all subsequent calls without going through its computation over and over again.

685a ⟨*Generic unparsing* 685a⟩≡
```
METHOD UnParse: Component;
```

685b ⟨*Generic unparsing implementation* 685b⟩≡
```
VAR
  TheUnParse: Component;

METHOD UnParse: Component;
  BEGIN
  IF TheUnParse = Λ THEN
    TheUnParse ← ComputeUnParse;
    END;
  RESULT ← TheUnParse;
  END UnParse;
```

The `ComputeUnParse` method is meant to be redefined by all classes deriving from `InterComponent`.

685c ⟨*Generic unparsing implementation* 685b⟩+≡
```
DEFERRED METHOD ComputeUnParse: Component;
```
YAFL

`GrabSubNodes` uses an exclusive feature of YAFL's type system to provide a dynamic form of selection of subnodes depending on their type.

An unconstrained list `l` is provided as parameter, and the purpose of this method is to visit all nodes of a subtree, and to store into `l` all elements which are compatible with `l`'s type. If `l` is a list of `InterExpression`, `GrabSubNodes` stores all `InterExpression` instances found in the subtree into `l`.

685d ⟨`GrabSubNodes` *method implementation* 685d⟩≡
```
METHOD GrabSubNodes (l: ArrayList);
  BEGIN
  ⟨First, check whether THIS should be stored in the list l 686a⟩
  -- Then, call GrabSubNodes recursively for all subnodes.
  FOR e ∈ SubTree | e ≠ Λ DO
    e.GrabSubNodes (l);
    END;
  END GrabSubNodes;
```

To check whether `THIS` is compatible with `l`'s elements, a WHAT control structure is used. `l.Element` denotes the types of the elements that can be stored in `l`, in accordance with the way `ArrayList` had been defined in the first place:

```
CLASS ArrayList(Element);
  ...
```

686a ⟨*First, check whether* `THIS` *should be stored in the list* `l` 686a⟩≡

```
WHAT THIS OF
  IN l.Element:
    l.Append (TAG);
    END;
 ELSE
  -- Don't abort
  END;
```

7.20.2 Intermediate types definition

`InterType` serves as base class for all type denotations that can occur within a lexical specification stored in this intermediate format, for instance, in relationship with an attribute.

686b ⟨*Intermediate types definition* 686b⟩≡
  ⟨`InterType` *base class for types, definition* 687a⟩
  ⟨*Predefined types, definition* 687c⟩
  ⟨*Nonterminal type* 691⟩
  ⟨`InterListType` *class definition* 696b⟩

`InterType` publishes two deferred methods. `GenerateMetaAttribute-Type` produces a YAFL type as part of a declaration such as `i: IntegerAttribute` for an attribute that will denote the *metaclass* of the class denoted by `THIS`.

For instance, if `THIS` denotes `InterBooleanType` (See page 688), `GenerateMetaAttributeType` will output `BooleanAttribute` (as provided by the YAFL standard library) on the output stream. If `THIS` denotes a user-defined class `Statement`, `GenerateMetaAttributeType` will output `ClassAttribute(Statement)`.

`GenerateFetch` is meant to be redefined by `InterType`-derived classes, and to produce code that extracts the `Index`th element from an array of `ParserStack-Element` (See page 332), and to fetch it into an attribute denoted by `TheAttribute`, assuming of course that this attribute's type matches THIS.

`InterType` also provides a method to test equality between types. At this primitive level, type equality is equivalent to reference equality. More sophisticated behaviours will be provided when `InterNonTerminalType` (See page 693) or `InterListType` (See page 701) redefine `IsEqual`.

687a ⟨`InterType` *base class for types, definition* 687a⟩≡

```
CLASS InterType;
  INHERITS InterComponent;

  DEFERRED METHOD GenerateMetaAttributeType
      (Output: DuraOutputStream);
  DEFERRED METHOD GenerateFetch
                      (Output: DuraOutputStream;
                          Index: INTEGER;
                          TheAttribute: InterAttribute);
  METHOD IsEqual (Other: InterType): BOOLEAN;
END InterType;
```

`InterType` does not perform much, I'm afraid.

687b ⟨`InterType` *base class for types, implementation* 687b⟩≡

```
CLASS InterType;
  INHERITS InterComponent;

  METHOD IsEqual (Other: InterType): BOOLEAN;
    BEGIN
    RESULT ← THIS = Other;
    END IsEqual;

END InterType;
```

ONCE classes are used to represent YAFL predefined types. The `REAL` and `CHAR` predefined types are not supported, since they serve no purpose in `DURA`'s context. Arrays are not supported, except for arrays of characters which are supported as a predefined type `InterStringType` (See page 689). YAFL

In `GenerateMetaAttributeType` as redefined by these three classes, it is worth mentioning that `IntegerAttribute`, `BooleanAttribute` and `StringAttribute` are classes provided by the YAFL standard library as part of the meta class system.

687c ⟨*Predefined types, definition* 687c⟩≡

```
ONCE CLASS InterIntegerType;
  INHERITS InterType;
END InterIntegerType;
```

688a ⟨*Predefined types, implementation* 688a⟩≡
```
ONCE CLASS InterIntegerType;
  INHERITS InterType;

  REDEFINE METHOD GenerateMetaAttributeType (Output: DuraOutputStream);
    BEGIN
    Output.WriteString (‘‘IntegerAttribute”);
    END GenerateMetaAttributeType;

  REDEFINE METHOD ComputeUnParse: Atom;
    BEGIN
    RESULT.CREATE(Λ, InteLex.IntegerType);
    END ComputeUnParse;

  REDEFINE METHOD SubTree: ARRAY OF InterComponent;
    BEGIN
    END SubTree;
```

Fetching an integer works by calling `ExtractInteger` (See page 880) as defined in the skeleton.

688b ⟨*Predefined types, implementation* 688a⟩+≡
```
REDEFINE METHOD GenerateFetch (Output: DuraOutputStream;
                               Index: INTEGER;
                               TheAttribute: InterAttribute);
  BEGIN
  TheAttribute.GenerateMetaAttribute (Output);
  Output.WriteString (‘‘.Set(ExtractInteger(TheArr[”);
  Output.WriteInt (Index, 0);
  Output.WriteLine (‘‘]));”);
  END GenerateFetch;

END InterIntegerType;
```

688c ⟨*Predefined types, definition* 687c⟩+≡
```
ONCE CLASS InterBooleanType;
  INHERITS InterType;
END InterBooleanType;
```

689a ⟨*Predefined types, implementation* 688a⟩+≡
```
ONCE CLASS InterBooleanType;
  INHERITS InterType;

  REDEFINE METHOD GenerateMetaAttributeType
                         (Output: DuraOutputStream);
    BEGIN
    Output.WriteString (‘‘BooleanAttribute”);
    END GenerateMetaAttributeType;

  REDEFINE METHOD ComputeUnParse: Atom;
    BEGIN
    RESULT.CREATE(Λ, InteLex.BooleanType);
    END ComputeUnParse;

  REDEFINE METHOD SubTree: ARRAY OF InterComponent;
    BEGIN
    END SubTree;
```

Deciding whether a `ParserStackElement` (See page 332) should fetch `TRUE` or `FALSE` is based on the `IsVoid` method (See page 878) as defined in the skeleton.

689b ⟨*Predefined types, implementation* 688a⟩+≡
```
REDEFINE METHOD GenerateFetch (Output: DuraOutputStream;
                               Index: INTEGER;
                               TheAttribute: InterAttribute);
  BEGIN
  TheAttribute.GenerateMetaAttribute (Output);
  Output.WriteString (‘‘.Set(NOT IsVoid(TheArr[”);
  Output.WriteInt (Index, 0);
  Output.WriteLine (‘‘]));”);
  END GenerateFetch;

END InterBooleanType;
```

689c ⟨*Predefined types, definition* 687c⟩+≡
```
ONCE CLASS InterStringType;
  INHERITS InterType;
END InterStringType;
```

690a ⟨*Predefined types, implementation* 688a⟩+≡
```
ONCE CLASS InterStringType;
  INHERITS InterType;

  REDEFINE METHOD GenerateMetaAttributeType (Output: DuraOutputStream);
    BEGIN
    Output.WriteString (‘‘StringAttribute”);
    END GenerateMetaAttributeType;

  REDEFINE METHOD ComputeUnParse: Atom;
    BEGIN
    RESULT.CREATE(Λ, InteLex.StringType);
    END ComputeUnParse;

  REDEFINE METHOD SubTree: ARRAY OF InterComponent;
    BEGIN
    END SubTree;
```

Extracting a character string from a `ParserStackElement` (See page 332) is performed by the `ExtractString` method (See page 881) as defined in the skeleton.

690b ⟨*Predefined types, implementation* 688a⟩+≡
```
REDEFINE METHOD GenerateFetch (Output: DuraOutputStream;
                               Index: INTEGER;
                               TheAttribute: InterAttribute);
  BEGIN
  TheAttribute.GenerateMetaAttribute (Output);
  Output.WriteString (‘‘.Set(ExtractString(TheArr[”);
  Output.WriteInt (Index, 0);
  Output.WriteLine (‘‘]));”);
  END GenerateFetch;

END InterStringType;
```

The `InterType`-derived classes listed above are ONCE classes since there is basically a single integer type, a single boolean type and a single character string type. On the other hand, `InterNonTerminalType` denotes a reference to a non-terminal class, and cannot be a ONCE class.

`InterNonTerminalType` supports two ways to refer to the class it describes:

- By a direct reference to a `DuraClassDefinition` (See page 415)
- By a character string built by concatenating the module name, a dot, and the class name.

Depending on whether we have the entire YAFL information available and we are currently producing the intermediate representation, or whether we have just read this intermediate representation, the first or the second of these ways of referring to the class will be used.

This duality is hidden from the user, all access methods will convert the information from one representation to the other on the fly if required.

691 ⟨*Nonterminal type* 691⟩≡

```
CLASS InterNonTerminalType;
  INHERITS InterType;

  METHOD QualId: ARRAY OF CHAR;
  METHOD SetQualId(QualId: ARRAY OF CHAR);
  METHOD Class: DuraClassDefinition;
  METHOD SetClass(Cl: DuraClassDefinition);
END InterNonTerminalType;
```

Well, based on what has been said above about the dual representation of the `InterNonTerminalType` class, the implementation is rather straightforward...

692 ⟨`InterNonTerminalType` *class implementation* 692⟩≡

```
CLASS InterNonTerminalType;
  INHERITS InterType;

  VAR
    TheQualId: ARRAY OF CHAR;
    TheClass: DuraClassDefinition;

  METHOD SetQualId (QualId: ARRAY OF CHAR);
    BEGIN
    TheQualId ← QualId;
    TheClass ← Λ;
    END SetQualId;

  ⟨Setting and fetching the attached class definition 693a⟩
  ⟨IsEqual, as redefined by InterNonTerminalType 693b⟩

  REDEFINE METHOD ComputeUnParse: Atom;
    BEGIN
    RESULT.CREATE(QualId, InteLex.QualId);
    END ComputeUnParse;

  REDEFINE METHOD SubTree: ARRAY OF InterComponent;
    BEGIN
    END SubTree;

  ⟨Generating the type for the meta-attribute of a InterNonTerminalType 694⟩
  ⟨Code generation to fetch a non-terminal attribute 696a⟩

END InterNonTerminalType;
```

The following methods manage the impedance mismatch between classes defined from within the YAFL compiler (as `DuraClassDefinition`, (See page 415)) and classes as saved in the intermediate format, as character string.

693a ⟨*Setting and fetching the attached class definition* 693a⟩≡

```
METHOD QualId: ARRAY OF CHAR;
  BEGIN
  IF TheClass ≠ Λ THEN
    RESULT ← TheClass.DotImage;
   ELSE
    RESULT ← TheQualId;
    END;
  END QualId;

METHOD SetClass(Cl: DuraClassDefinition);
  BEGIN
  TheQualId ← Λ;
  TheClass ← Cl;
  END SetClass;

METHOD Class: DuraClassDefinition;
  BEGIN
  IF TheClass = Λ THEN
    TheClass ← DuraCfg.GetClass (TheQualId);
    END;
  RESULT ← TheClass;
  END Class;
```

`InterNonTerminalType` extends the type equality method as defined by its base class (See page 686) to support the ability to refer to the same non-terminal class as a way (beside reference equality) of denoting the same type.

693b ⟨`IsEqual`, *as redefined by* `InterNonTerminalType` 693b⟩≡

```
REDEFINE METHOD IsEqual (Other: InterType): BOOLEAN;
  BEGIN
  RESULT ← BASE(Other);
  IF ¬ RESULT THEN
    WHAT Other OF
      IN InterNonTerminalType:
        RESULT ← Class = TAG.Class;
        END;
     ELSE
      END;
    END;
  END IsEqual;
```

The YAFL standard library defines a meta-class for each of the simple types (integer, character, boolean and real). When one must describe a user-defined type, one must use the parameterized class `ClassAttribute`, constrained by the class one must describe. In other word, an attribute that refers to a polygon would be denoted by `ClassAttribute(Polygon)` in the YAFL meta-class system. YAFL

694 ⟨*Generating the type for the meta-attribute of a* `InterNonTerminalType` 694⟩≡

```
REDEFINE METHOD GenerateMetaAttributeType
                      (Output: DuraOutputStream);
  BEGIN
  Output.WriteString (``ClassAttribute(" +
                        Class.YaflClassName +
                          ``)");
  END GenerateMetaAttributeType;
```

YAFL

Instead of explaining the tedious steps of the method below, the behaviour of this code generation method will be listed in an example.

Let's consider an attribute of the `Statement` class, for which the corresponding meta attribute (See method *DuraIntermediateFormat.InterAttribute.GenerateMetaAttribute* on page 738) is called `A_504`. The code DURA generates to fetch this arribute from the second position in the array `TheArr` goes as follows:

```
p := ExtractNonTerminal(TheArr[2]);
IF p = VOID THEN
  A_504.Set(VOID);
 ELSE
  WHAT p OF
    IN Statement:
      A_504.Set(TAG);
      Nt.SetSon(TAG);
      END;
    END;
  END;
```

Several comments can be made about this chunk of code:

- `ExtractNonTerminal` (See page 882) returns a `BaseNonTerminal` (See page 346) instance. One must use the `WHAT` construct to constrain it to a `Statement`.
- This `WHAT` statement might fail if the original grammar rule with its attached attribute assignment is ill-formed, that is, if one attempts to assign a `Statement` onto an attribute that cannot refer to a `Statement`. This check only happens at runtime, while it could be performed statically by `DURA` when generating the parser or even, as early as when the intermediate representation is generated for the first time. In practice, this programming error seldom occurs, and this dynamic detection scheme, far easier to develop than an equivalent static scheme, has proven to be good enough for all practical purposes.
- `Nt` refers to the non-terminal which is currently being reduced. When the `Statement` is fetched into the proper attribute, it is marked as being one of `Nt`'s children, by calling the `SetSon` method (See page 357).
- As the example above shows, `GenerateFetch` produces properly indented code. This has been achieved by using a `DuraOutputStream` (See page 859) instead of a plain `OutputStream`. `DuraOutputStream` handles indentation, and various conveniency methods such as `End` (See page 861) that outputs the "`END`" keyword, and then decreases the indent level by 2.

696a ⟨*Code generation to fetch a non-terminal attribute* 696a⟩≡

```
REDEFINE METHOD GenerateFetch (Output: DuraOutputStream;
                               Index: INTEGER;
                               TheAttribute: InterAttribute);
  BEGIN
  Output.WriteString (“p := ExtractNonTerminal(TheArr[”);
  Output.WriteInt (Index, 0);
  Output.WriteLine (“]);”);
  Output.WriteLine (“IF p = VOID THEN”);
  Output.IncIndent (2);
  TheAttribute.GenerateMetaAttribute (Output);
  Output.WriteLine (“.Set(VOID);”);
  Output.Else;
  Output.WriteLine (“WHAT p OF”);
  Output.IncIndent (2);
  Output.WriteLine (“IN ” + Class.YaflClassName + “:”);
  Output.IncIndent (2);
  TheAttribute.GenerateMetaAttribute (Output);
  Output.WriteLine (“.Set(TAG);”);
  Output.WriteLine (“Nt.SetSon(TAG);”);
  Output.End;
  Output.End;
  Output.End;
  END GenerateFetch;
```

Besides assigning non-terminals to attributes, one can also assign a *list* of such non-terminals to an attribute denoted by the `NonTerminalList` class (See page 360). `InterListType` describes the type of such attributes in the intermediate representation. It is of course not sufficient to just indicate we are dealing with lists, since the actual class can be derived from `NonTerminalList`. We also have to hold information about possible parameters.

We shall refer to the actual class (derived from `NonTerminalList`) as the *true class.*

696b ⟨`InterListType` *class definition* 696b⟩≡

```
CLASS InterListType;
  INHERITS InterType;

  METHOD TrueClass: DuraClassDefinition;
  METHOD SetTrueClass (TrueClass: DuraClassDefinition);
  METHOD AppendActual (Act: DuraClassDefinition);

  METHOD GenerateCreationMethodBody
                   (Output: DuraOutputStream);

END InterListType;
```

The implementation is rather straightforward.

697a ⟨`InterListType` *class implementation* 697a⟩≡

```
CLASS InterListType;
  INHERITS InterType;

  VAR
    TheTrueClass: DuraClassDefinition;
    TheActuals: NTList(DuraClassDefinition);

  ⟨Consulting and setting the attached class 697b⟩
  ⟨Consulting and setting the attached actual classes 697c⟩
  ⟨Unparsing an InterListType 698a⟩
  ⟨Code generation methods as redefined by InterListType 698b⟩
  ⟨Checking whether two InterListType instances are equal 702a⟩

END InterListType;
```

697b ⟨*Consulting and setting the attached class* 697b⟩≡

```
METHOD TrueClass: DuraClassDefinition;
  BEGIN
  RESULT ← TheTrueClass;
  END TrueClass;

METHOD SetTrueClass (TrueClass: DuraClassDefinition);
  BEGIN
  TheTrueClass ← TrueClass;
  END SetTrueClass;
```

697c ⟨*Consulting and setting the attached actual classes* 697c⟩≡

```
METHOD AppendActual (Act: DuraClassDefinition);
  BEGIN
  IF TheActuals = Λ THEN
    TheActuals.CREATE;
    END;
  TheActuals.Append (Act);
  END AppendActual;

METHOD Actuals: NTList(DuraClassDefinition);
  BEGIN
  RESULT ← TheActuals;
  END Actuals;
```

Unparsing an `InterListType` yields a `Compound` (See page 801), that is, a list of items as opposed to an `Atom` (See page 798). The original `ComputeUnParse` method (See page 685) is defined as returning a `Component` (See page 797) which is the base class for both `Atom` and `Compound`.

YAFL

We make use of *constrained redefinition* as provided by YAFL, which allows us to redefine a method returning an instance of a given class C with a method that returns an instance of a class D derived from C. [31] explains the limitations of this scheme, and why it cannot be extended to the method's formal parameters.

698a ⟨*Unparsing an* InterListType 698a⟩≡

```
REDEFINE METHOD ComputeUnParse: Compound;
  BEGIN
  RESULT.CREATE;
  RESULT.AppendAtom (InteLex.ListType);
  RESULT.AppendAtomText (InteLex.QualId, TrueClass.DotImage);
  FOR Act ∈ Actuals DO
    RESULT.AppendAtomText (InteLex.QualId, Act.DotImage);
    END;
  END ComputeUnParse;
```

The motto for this redefinition of GenerateMetaAttributeType is the same that was described for InterNonTerminalType (See page 694).

698b ⟨*Code generation methods as redefined by* InterListType 698b⟩≡

```
REDEFINE METHOD GenerateMetaAttributeType
                          (Output: DuraOutputStream);
  BEGIN
  Output.WriteString (``ClassAttribute(" +
                        TrueClass.YaflClassName + ``)");
  END GenerateMetaAttributeType;
```

Most of the rationale behind the redefinition of GenerateFetch for InterList-Type is similar to what is described for InterNonTerminalType. We consider an attribute that holds a list of statements, for which the corresponding meta attribute (See method *DuraIntermediateFormat.InterAttribute.GenerateMetaAttribute* on page 738) is called A_721, GenerateFetch produces an output such as:

```
p := ExtractNonTerminal(TheArr[1]);
IF p = VOID THEN
  A_721.Set(VOID);
 ELSE
  A_721.Set(C_178_4182);
  FetchList(A_721.Get, p);
  IF A_721.Get.Size = 0 THEN
    A_721.Set(VOID);
   ELSE
    Nt.SetSons(A_721.Get);
    END;
  END;
```

In the example above, `C_178_4182` is a method generated as part of the skeleton and which is reponsible for creating a new instance of the list type denoted by `THIS`.

- `GenerateFetch` first extracts the *last non-terminal* of the sequence by calling `ExtractNonTerminal` (See page 882) as defined in the skeleton
- If such a non-terminal exists, a list is created (See method *DuraIntermediateFormat.InterListType.GenerateCreationMethodName* on the next page) and set in the corresponding meta-attribute.
- `GenerateFetch` then calls `FetchList` (See page 883) to accumulate the predecessors of this last non-terminal in the newly created list
- If the resulting list is empty, the attribute is reset to void[5]. This ensures us that we never have to deal with the case of a `CREATE`d but empty list. An empty list is always denoted by the `VOID` value. This requires less memory, and is consistent with a form of convention throughout the YAFL compiler and libraries.
- `Nt` refers to the non-terminal which is currently being reduced. When the list is fetched into the proper attribute, it is marked as made of `Nt`'s children, by calling the `SetSons` method (See page 357).

699 ⟨*Code generation methods as redefined by* `InterListType` 698b⟩+≡

```
REDEFINE METHOD GenerateFetch (Output: DuraOutputStream;
                               Index: INTEGER;
                               TheAttribute: InterAttribute);
  BEGIN
  Output.WriteString (“p := ExtractNonTerminal(TheArr[”);
  Output.WriteInt (Index, 0);
  Output.WriteLine (“]);”);
  Output.WriteLine (“IF p = VOID THEN”);
  Output.IncIndent (2);
  TheAttribute.GenerateMetaAttribute (Output);
  Output.WriteLine (“.Set(VOID);”);
  Output.Else;
  TheAttribute.GenerateMetaAttribute (Output);
  Output.WriteString (“.Set(”);
  GenerateCreationMethodName (Output);
  Output.WriteLine (“);”);
  Output.WriteString (“FetchList(”);
  TheAttribute.GenerateMetaAttribute (Output);
  Output.WriteLine (“.Get, p);”);
  Output.WriteString (“IF ”);
  TheAttribute.GenerateMetaAttribute (Output);
  Output.WriteLine (“.Get.Size = 0 THEN”);
```

[5] In theory, this should not be necessary, since we know at this stage that there is at least a single element to be stored in the list. Making this redundant check is a result of the paranoia one can develop after being in the software industry for over a decade.

```
    Output.IncIndent (2);
    TheAttribute.GenerateMetaAttribute (Output);
    Output.WriteLine (‘‘.Set(VOID);”);
    Output.Else;
    Output.WriteString (‘‘Nt.SetSons(”);
    TheAttribute.GenerateMetaAttribute (Output);
    Output.WriteLine (‘‘.Get);”);
    Output.End;
    Output.End;
    END GenerateFetch;
```

YAFL

The identifier produced on `Output` by `GenerateCreationMethodName` is built using hash values that are attached to all non-terminals by the YAFL compiler at parse time. The value of these hash values is guaranteed to remain valid within a single compilation run, but not beyond. When parsing the same module in another run of the YAFL compiler, the hash values can be different.

In our very specific case, this is no serious limitation, since we use this hash value when producing the parser, in a single run of the – significantly extended – YAFL compiler. If we generated a *modular* parser piece by piece instead of a monolithic one, this would be more of a concern.

700 ⟨*Code generation methods as redefined by* `InterListType` 698b⟩+≡

```
  METHOD GenerateCreationMethodName (Output: DuraOutputStream);
    BEGIN
    Output.WriteString(‘‘CL_”);
    Output.WriteInt (TrueClass.HashValue, 0);
    FOR Act ∈ TheActuals DO
      Output.WriteChar (‘‘_”);
      Output.WriteInt (Act.HashValue, 0);
      END;
    END GenerateCreationMethodName;
```

`GenerateCreationMethodBody` generates an entire method that will create an instance of the list type denoted by `THIS`. For instance, if `THIS` denotes a type `StatList(IfStatement)` (assuming of course that `StatList` inherits from `NonTerminalList` (See page 360)), `GenerateCreationMethodBody` could generate something such as:

```
METHOD CL_1072_91: StatList(IfStatement);
  BEGIN
  RESULT.CREATE;
  END CL_1072_91;
```

For this to be possible, it is of course essential that the various `NonTerminalList`-derived classes one defines support a parameterless `CREATE` method.

701 ⟨*Code generation methods as redefined by* `InterListType` 698b⟩+≡

```
METHOD GenerateCreationMethodBody (Output: DuraOutputStream);
  BEGIN
  Output.WriteString (“METHOD ”);
  GenerateCreationMethodName (Output);
  Output.WriteString (“ : ”);
  Output.WriteString (TrueClass.YaflClassName);
  IF (TheActuals ≠ Λ) THEN
    ASSERT TheActuals.Size > 0;
    Output.WriteChar (“(”);
    FOR i ∈ 0 .. TheActuals.Size - 1 DO
      IF i > 0 THEN
        Output.WriteString (“, ”);
        END;
      Output.WriteString (TheActuals_i;
      END;
    Output.WriteChar (“)”);
    END;
  Output.WriteLine (“;”);
  Output.IncIndent (2);
  Output.WriteLine (“BEGIN”);
  Output.WriteLine (“RESULT.CREATE;”);
  Output.WriteString (“END ”);
  GenerateCreationMethodName (Output);
  Output.WriteLine (“;”);
  Output.DecIndent (2);
  Output.WriteLn;
  END GenerateCreationMethodBody;
```

`IsEqual` is a utility method that simply checks whether two `InterListType` instances are equal beyond the reference-based equality which is provided by the `InterType` base class. Thanks to YAFL's compiler structure where each class and type is only represented once, this equality can be based on references when it comes to ensuring that two YAFL class references are the same. YAFL

702a ⟨*Checking whether two* `InterListType` *instances are equal* 702a⟩≡
```
  REDEFINE METHOD IsEqual (Other: InterType): BOOLEAN;
    BEGIN
    RESULT ← BASE(Other);
    IF ¬ RESULT THEN
      WHAT Other OF
        IN InterListType:
          ASSERT TrueClass ≠ Λ;
          ASSERT TAG.TrueClass ≠ Λ;
          IF TrueClass = TAG.TrueClass THEN
            RESULT ← (Actuals = Λ) ∨
                       (∀ i ∈ 0 .. Actuals.Size - 1 •
                                   Actuals_i = TAG.Actuals_i;
            END;
          END;
        END;
     ELSE
      END;
    END IsEqual;
```

7.20.3 Representing grammatical expressions at the intermediate level

702b ⟨*Intermediate expression class definitions* 702b⟩≡
- ⟨`InterExpression` *class definition* 703b⟩
- ⟨`InterAttribute` *class definition* 735a⟩
- ⟨`InterThis` *class definition* 740c⟩
- ⟨`InterAggregateExpression` *class definition* 706⟩
- ⟨`InterTerminalExpression` *class definition* 710a⟩
- ⟨`InterTerminalSetExpression` *class definition* 712a⟩
- ⟨`InterNonTerminalExpression` *class definition* 715b⟩
- ⟨`InterOptionalExpression` *class definition* 718b⟩
- ⟨`InterRepetitionExpression` *class definition* 720b⟩
- ⟨`InterCaseExpression` *class definition* 723c⟩
- ⟨`InterCaseAlt` *class definition* 726b⟩

703a ⟨*Intermediate expression class implementation* 703a⟩≡

```
⟨InterAttribute class implementation 735b⟩
⟨InterThis class implementation 741a⟩
⟨InterExpression class implementation 703c⟩
  ⟨InterAggregateExpression class implementation 707⟩
  ⟨InterTerminalExpression class implementation 710b⟩
  ⟨InterTerminalSetExpression class implementation 712b⟩
  ⟨InterNonTerminalExpression class implementation 716⟩
  ⟨InterOptionalExpression class implementation 719⟩
  ⟨InterRepetitionExpression class implementation 721⟩
  ⟨InterCaseExpression class implementation 724⟩
  ⟨InterCaseAlt class implementation 727⟩
```

`InterExpression` serves as base class for all grammatical expression classes listed in this module. It basically provides two generic (or redefinable) services:

- Setting and reading a target attribute in which the parse tree produced by the grammatical expression will be fetched
- Giving the list of attributes that are attached (as target for assignments) by this expression or any of its sub-expressions

703b ⟨`InterExpression` *class definition* 703b⟩≡

```
CLASS InterExpression;
  INHERITS InterComponent;

  ⟨Target management, definition 703d⟩

  ⟨RelatedAttributes method definition 705c⟩

END InterExpression;
```

703c ⟨`InterExpression` *class implementation* 703c⟩≡

```
CLASS InterExpression;
  INHERITS InterComponent;

  ⟨Target management, implementation 704a⟩
  ⟨Deferred methods to be redefined by InterExpression derived classes 704b⟩
  ⟨Returning an equivalent FlatRHS 705a⟩
  ⟨RelatedAttributes method implementation 705d⟩

  ⟨Unparsing an InterExpression 704c⟩

END InterExpression;
```

703d ⟨*Target management, definition* 703d⟩≡

```
METHOD SetTarget (Target: InterAttribute);
METHOD Target: InterAttribute;
```

704a ⟨*Target management, implementation* 704a⟩≡
```
VAR
  TheTarget: InterAttribute;

METHOD SetTarget (Target: InterAttribute);
  BEGIN
  TheTarget ← Target;
  END SetTarget;

METHOD Target: InterAttribute;
  BEGIN
  RESULT ← TheTarget;
  END Target;
```

704b ⟨*Deferred methods to be redefined by* InterExpression *derived classes* 704b⟩≡
```
DEFERRED METHOD Parse(Comp: Compound);
```

ComputeUnParse was already part of a two level scaffolding for the unparsing of an InterComponent (See page 685) to be performed lazily. We add another level here. ComputeUnParse calls UnParse (See this page) and if a target attribute has been defined, encloses the resulting Compound in another one that includes this assignment information.

704c ⟨*Unparsing an* InterExpression 704c⟩≡
```
REDEFINE METHOD ComputeUnParse: Compound;
  BEGIN
  IF Target = Λ THEN
    RESULT ← UnParseValue;
   ELSE
    RESULT.CREATE;
    RESULT.AppendAtom (InteLex.Assign);
    RESULT.Append (Target.UnParse);
    RESULT.Append (UnParseValue);
    END;
  END ComputeUnParse;
```

704d ⟨*Deferred methods to be redefined by* InterExpression *derived classes* 704b⟩+≡
```
DEFERRED METHOD UnParseValue: Compound;
```

FlatReference basically calls BuildFlatReference (See page 728) and attaches the target attribute if defined.

It returns an element that refers to the grammatical expression denoted by THIS, that can be used in the flattened representation (See BuildFlatRHS below).

705a ⟨*Returning an equivalent* FlatRHS 705a⟩≡
```
METHOD FlatReference: FlatRuleComponent;
  BEGIN
  RESULT ← BuildFlatReference;
  IF RESULT ≠ Λ THEN
    RESULT.SetTarget (Target);
    END;
  END FlatReference;
```

705b ⟨*Returning an equivalent* FlatRHS 705a⟩+≡
```
METHOD BuildFlatRHS: FlatRHS;
  VAR
    Component: FlatRuleComponent;
  BEGIN
  RESULT.CREATE;
  Component ← FlatReference;
  IF Component ≠ Λ THEN
    RESULT.Append (Component);
    END;
  END BuildFlatRHS;
```

The RelatedAttributes method returns an array of attributes required to fetch the expression in THIS.

705c ⟨RelatedAttributes *method definition* 705c⟩≡
```
METHOD RelatedAttributes: ARRAY OF InterAttribute;
```

In practice, there are two cases to consider:

- A target attribute has been attached to the expression denoted by THIS
- In case of an InterCaseExpression (See page 723), the attribute that receives the discriminant value will be considered within a redefinition of this method. This latter case will be checked by an ad hoc redefinition (See page 726).

705d ⟨RelatedAttributes *method implementation* 705d⟩≡
```
METHOD RelatedAttributes: ARRAY OF InterAttribute;
  BEGIN
  IF (Target ≠ Λ) ∧ (Target ≠ InterThis) THEN
    RESULT.CREATE (1);
    RESULT₀ ← Target;
    END;
  END RelatedAttributes;
```

7.20.3.1 Aggregates

The `InterAggregateExpression` represents aggregate expressions of two kinds: concatenations and disjunctions.

706 ⟨`InterAggregateExpression` *class definition* 706⟩≡

```
CLASS InterAggregateExpression;
  INHERITS InterExpression;

  CONST
    ConcatOp = 978;
    DisjunctionOp = 979;

  REDEFINE METHOD CREATE;

  METHOD SetOp (Op: INTEGER);
    PRE
      (Op = ConcatOp) ∨ (Op = DisjunctionOp);

  METHOD Op: INTEGER;
    POST
      (RESULT = ConcatOp) ∨ (RESULT = DisjunctionOp);

  METHOD Append (Expr: InterExpression);
  METHOD Args: ArrayList(InterExpression);

END InterAggregateExpression;
```

This class is naive, except for the way it handles the flat representation (See method *DuraIntermediateFormat.InterAggregateExpression.BuildFlatReference* on page 730).

707 ⟨InterAggregateExpression *class implementation* 707⟩≡

```
CLASS InterAggregateExpression;
  INHERITS InterExpression;

  VAR
    TheOp: INTEGER;
    TheArgs: ArrayList(InterExpression);

  REDEFINE METHOD CREATE;
    BEGIN
    BASE;
    TheArgs.CREATE;
    END CREATE;

  ⟨Naive attribute management for InterAggregateExpression 708⟩
  ⟨Parsing and unparsing an InterAggregateExpression 709a⟩

  REDEFINE METHOD SubTree: ARRAY OF InterComponent;
    BEGIN
    RESULT.CREATE (Args.Size);
    FOR i ∈ 0 .. RESULT.SIZE - 1 DO
      RESULT_i ← Args_i;
      END;
    END SubTree;

  ⟨BuildFlatReference, as redefined by InterAggregateExpression 730a⟩

END InterAggregateExpression;
```

Pretty boring stuff...

708 ⟨*Naive attribute management for* `InterAggregateExpression` 708⟩≡

```
METHOD SetOp (Op: INTEGER);
  BEGIN
  ASSERT (Op = ConcatOp) ∨ (Op = DisjunctionOp);
  TheOp ← Op;
  END SetOp;

METHOD Op: INTEGER;
  BEGIN
  RESULT ← TheOp;
  ASSERT (RESULT = ConcatOp) ∨ (RESULT = DisjunctionOp);
  END Op;

METHOD Append (Expr: InterExpression);
  BEGIN
  TheArgs.Append (Expr);
  END Append;

METHOD Args: ArrayList(InterExpression);
  BEGIN
  RESULT ← TheArgs;
  END Args;
```

Representing an aggregate in the list form (that is, as a `Compound` (See page 801)) is pretty simple: the first element indicates whether we are dealing with a disjunction or with a concatenation, and the other elements are made of the arguments, turned into `Compound`'s themselves by calling `UnParse` (See page 685), and turned back into the intermediate representation by calling `ParseExpression`, as defined by `IntermediateParser` (See page 759).

709a ⟨*Parsing and unparsing an* InterAggregateExpression 709a⟩≡
```
REDEFINE METHOD Parse(Comp: Compound);
  BEGIN
  CASE Comp0.AtomCode OF
    InteLex.Or:
      SetOp (DisjunctionOp);
      END;
    InteLex.Concat:
      SetOp (ConcatOp);
      END;
    END;
  FOR i ∈ 1 .. Comp.Size-1 DO
    Append (IntermediateParser.
            ParseExpression(Compi;
    END;
  END Parse;
```

709b ⟨*Parsing and unparsing an* InterAggregateExpression 709a⟩+≡
```
REDEFINE METHOD UnParseValue: Compound;
  BEGIN
  RESULT.CREATE;
  CASE Op OF
    DisjunctionOp:
      RESULT.AppendAtom(InteLex.Or);
      END;
    ConcatOp:
      RESULT.AppendAtom(InteLex.Concat);
      END;
    END;
  FOR a ∈ Args DO
    RESULT.Append(a.UnParse);
    END;
  END UnParseValue;
```

7.20.3.2 InterExpression-derived classes

InterTerminalExpression denotes a grammatical expression that matches a single token. The various methods provided by this class here are pretty straightforward.

710a ⟨InterTerminalExpression *class definition* 710a⟩≡

```
CLASS InterTerminalExpression;
  INHERITS InterExpression;

  METHOD Set(LexCode: INTEGER;
             TokenName: ARRAY OF CHAR);

  METHOD LexCode: INTEGER;
    POST
     NO_CHANGE;

  METHOD TokenName: ARRAY OF CHAR;

END InterTerminalExpression;
```

710b ⟨InterTerminalExpression *class implementation* 710b⟩≡

```
CLASS InterTerminalExpression;
  INHERITS InterExpression;

  ⟨Setting and consulting the lexical constant attached to THIS 711a⟩
  ⟨Parsing and unparsing an InterTerminalExpression 711c⟩

  REDEFINE METHOD SubTree: ARRAY OF InterComponent;
    BEGIN
    END SubTree;

  ⟨BuildFlatReference as redefined by InterTerminalExpression 731b⟩

END InterTerminalExpression;
```

711a ⟨*Setting and consulting the lexical constant attached to* `THIS` 711a⟩≡
```
VAR
  TheTokenName: ARRAY OF CHAR;
  TheLexCode: INTEGER;

METHOD LexCode: INTEGER;
  BEGIN
  RESULT ← TheLexCode;
  END LexCode;

METHOD Set(LexCode: INTEGER;
           TokenName: ARRAY OF CHAR);
  BEGIN
  TheLexCode ← LexCode;
  TheTokenName ← TokenName;
  END Set;
```

If the token name has not been set, we use `LexemeImage` (See page 379) to turn the code into a character string representation.

711b ⟨*Setting and consulting the lexical constant attached to* `THIS` 711a⟩+≡
```
METHOD TokenName: ARRAY OF CHAR;
  BEGIN
  IF TheTokenName = Λ THEN
    RESULT ← DuraCfg.LexemeImage (LexCode);
   ELSE
    RESULT ← TheTokenName;
    END;
  END TokenName;
```

711c ⟨*Parsing and unparsing an* `InterTerminalExpression` 711c⟩≡
```
REDEFINE METHOD Parse(Comp: Compound);
  VAR
    a: ARRAY OF CHAR;
  BEGIN
  ASSERT Comp_0.AtomCode = InteLex.Terminal;
  ASSERT Comp_1.AtomCode = InteLex.Ident;
  a ← Space.StoreString(Comp_1;
  Set(DuraCfg.FindLexeme (a), a);
  END Parse;

REDEFINE METHOD UnParseValue: Compound;
  BEGIN
  RESULT.CREATE;
  RESULT.AppendAtom(InteLex.Terminal);
  RESULT.AppendAtomText(InteLex.Ident, TokenName);
  END UnParseValue;
```

InterTerminalSetExpression is a conceptual[6] generalization of the InterTerminalExpression, that matches not just *one* given token, but one of a *set* of tokens.

712a ⟨InterTerminalSetExpression *class definition* 712a⟩≡
```
CLASS InterTerminalSetExpression;
  INHERITS InterExpression;

  METHOD SetTerminalSet(TermSet: TerminalSet);

END InterTerminalSetExpression;
```

Internally, the set of tokens is denoted by the TheSet attribute (See page 558).

712b ⟨InterTerminalSetExpression *class implementation* 712b⟩≡
```
CLASS InterTerminalSetExpression;
  INHERITS InterExpression;

  VAR
    TheSet: TerminalSet;

  METHOD SetTerminalSet(TermSet: TerminalSet);
    BEGIN
    TheSet ← TermSet;
    END SetTerminalSet;

  ⟨Parsing and unparsing an InterTerminalSetExpression 713⟩

  REDEFINE METHOD SubTree: ARRAY OF InterComponent;
    BEGIN
    END SubTree;

  ⟨BuildFlatReference as redefined by InterTerminalSetExpression 731c⟩

END InterTerminalSetExpression;
```

Representing a set of tokens as a list can be made very simple, by just enumerating all the tokens one by one. However, this naive scheme would be inefficient in terms of storage. For instance, we now have a COBOL grammar that includes about 500 different token codes, and where we have grammar rules such as [* - EndExec] that matches any token except the END-EXEC token.

In such cases, listing all the possible tokens verbatim would of course be a little tedious and inefficient. In order to address this problem, when unparsing[7] such a set, we detected *contiguous ranges* of tokens that are in the token set, and represent them using a compact notation.

[6] As opposed to a *concrete* generalization that would turn into some inheritance relationship

[7] Understand: turning into a list

In practice, an `InterTerminalSetExpression`, when turned into a list, starts with a code that indicates its natures, followed by `Atom`'s for loose elements and lists for intervals. These lists that describe intervals are made of three elements themselves: the first element indicates its nature (that is, a range), followed by the first and the last element of the range. In this representation, all references to tokens are made through their character string representation rather than their integer counterparts. This is achieved by the `FindLexeme` method (See page 377) to turn a character string into the matching code; and the `LexemeImage` method (See page 379) for the opposite operation.

713 ⟨*Parsing and unparsing an* `InterTerminalSetExpression` 713⟩≡

```
REDEFINE METHOD Parse(Comp: Compound);
  VAR
    FromCode, ToCode, Code: INTEGER;
  BEGIN
  ASSERT Comp_0.AtomCode = InteLex.TerminalSet;
  TheSet.CREATE;
  FOR i ∈ 1 .. Comp.Size - 1 DO
    WHAT Comp_i OF
      IN Atom:
        Code ← DuraCfg.FindLexeme(TAG.Text);
        IF Code ≥ 0 THEN
          TheSet.Set(Code);
         ELSE
          DEBUG
            StdErr.WriteLine (“Unrecognized token: ” + TAG.Text);
            END;
          END;
        END;
      IN Compound:
        ASSERT TAG_0.AtomCode = InteLex.Range;
        FromCode ← SYSTEM.ABS(DuraCfg.FindLexeme(TAG_1.
                                          CastToAtom.Text));
        ToCode ← SYSTEM.ABS(DuraCfg.FindLexeme(TAG_2.
                                        CastToAtom.Text));
        ASSERT (FromCode ≥ 0) ∧ (ToCode ≥ 0);
        ASSERT ToCode ≥ FromCode;
        FOR j ∈ FromCode .. ToCode DO
          TheSet.Set(j);
          END;
        END;
      END;
    END;
  END Parse;
```

Unparsing this range-based list representation requires that some intelligence is put into it to actually *detect* such intervals. The following chunk maintains a current range, considering every token in the terminal set in sequence, extending the range when possible, and *flushing* it when the range cannot be extended any further.

714a ⟨*Parsing and unparsing an* `InterTerminalSetExpression` 713⟩+≡

```
REDEFINE METHOD UnParseValue: Compound;
  VAR
    CurStart, Cursor: INTEGER;
    Res: Compound;

    ⟨Local method Flush 715a⟩

  BEGIN
  ⟨Set CurStart and Cursor to nil values 714b⟩
  Res.CREATE;
  Res.AppendAtom(InteLex.TerminalSet);
  FOR i ∈ DuraCfg.FirstToken .. DuraCfg.LastToken | TheSet_i DO
    IF i = Cursor + 1 THEN
      Cursor ← i;
     ELSE
      Flush;
      CurStart ← i;
      Cursor ← i;
      END;
    END;
  Flush;
  RESULT ← Res;
  END UnParseValue;
```

The two variable are set to negative values to indicate that there is no interval currently being defined.

714b ⟨*Set* `CurStart` *and* `Cursor` *to nil values* 714b⟩≡

```
CurStart ← -8;
Cursor ← -8;
```

Flushing the current interval is merely a matter of deciding whether we will append a list that describes a range or append one or more atomic elements. Quite arbitrarily, we consider that all ranges greater than a threshold (currently set to 3) ought to be represented by lists.

715a ⟨*Local method* Flush 715a⟩≡

```
METHOD Flush;
  VAR
    Temp: Compound;
  BEGIN
  IF CurStart ≥ 0 THEN
    ASSERT Cursor ≥ CurStart;
    IF Cursor - CurStart > 3 THEN
      Temp.CREATE;
      Temp.AppendAtom (InteLex.Range);
      Temp.AppendAtomText (InteLex.Ident,
                           DuraCfg.LexemeImage(CurStart));
      Temp.AppendAtomText (InteLex.Ident,
                           DuraCfg.LexemeImage(Cursor));
      Res.Append (Temp);
     ELSE
      FOR i ∈ CurStart .. Cursor DO
        Res.AppendAtomText (InteLex.Ident, DuraCfg.LexemeImage(i));
        END;
      END;
    END;
  ⟨Set CurStart and Cursor to nil values 714b⟩
  END Flush;
```

InterNonTerminalExpression denotes a grammatical expression that refers to a non-terminal class. These non-terminal classes are YAFL classes, not just any synthetic non-terminal defined in the flattened representation.

715b ⟨InterNonTerminalExpression *class definition* 715b⟩≡

```
CLASS InterNonTerminalExpression;
  INHERITS InterExpression;

  METHOD SetQualId (QualId: ARRAY OF CHAR);
  METHOD QualId: ARRAY OF CHAR;
  METHOD Class: DuraClassDefinition;
  METHOD SetClass(Cl: DuraClassDefinition);

END InterNonTerminalExpression;
```

For pragmatic reasons, the class the `InterNonTerminalExpression` refers to is managed in a dual way, by maintaining both a character string (a `QualId`, using the *Module.Class* notation), and a direct reference to the class (See class *DuraClDefinition.DuraClassDefinition* on page 415).

This duality is mostly maintained internally, in such a way that when one of the two possible views on this information is missing, the other one is used.

716 ⟨`InterNonTerminalExpression` *class implementation* 716⟩≡

```
CLASS InterNonTerminalExpression;
  INHERITS InterExpression;

  ⟨Managing InterNonTerminalExpression's dual nature 717a⟩

  REDEFINE METHOD Parse(Comp: Compound);
    BEGIN
    ASSERT Comp.Size ≥ 2;
    ASSERT Comp0.AtomCode = InteLex.NonTerminal;
    ASSERT Comp1.AtomCode = InteLex.QualId;
    SetQualId(Comp1;
    END Parse;

  REDEFINE METHOD UnParseValue: Compound;
    BEGIN
    RESULT.CREATE;
    RESULT.AppendAtom(InteLex.NonTerminal);
    RESULT.AppendAtomText(InteLex.QualId, QualId);
    END UnParseValue;


  ⟨SubTree as redefined by InterNonTerminalExpression 717c⟩
  ⟨BuildFlatReference as redefined by InterNonTerminalExpression 731d⟩
  ⟨InterNonTerminalExpression semantic analysis 718a⟩

END InterNonTerminalExpression;
```

717a ⟨*Managing* `InterNonTerminalExpression`*'s dual nature* 717a⟩≡
```
VAR
  TheQualId: ARRAY OF CHAR;
  TheClass: DuraClassDefinition;

METHOD SetQualId (QualId: ARRAY OF CHAR);
  BEGIN
  TheQualId ← QualId;
  TheClass ← Λ;
  END SetQualId;

METHOD QualId: ARRAY OF CHAR;
  BEGIN
  RESULT ← TheQualId;
  END QualId;

METHOD SetClass (Cl: DuraClassDefinition);
  BEGIN
  TheClass ← Cl;
  TheQualId ← Cl.DotImage;
  END SetClass;
```

When asking for the actual class `THIS` refers to, if the corresponding attribute is not set, we use the `GetClass` method (See page 375) and the character string representation.

717b ⟨*Managing* `InterNonTerminalExpression`*'s dual nature* 717a⟩+≡
```
METHOD Class: DuraClassDefinition;
  BEGIN
  IF TheClass = Λ THEN
    TheClass ← DuraCfg.GetClass (TheQualId);
    END;
  RESULT ← TheClass;
  END Class;
```

In order to integrate with the YAFL compiler, `SubTree` must be redefined, even if, as in the case here, there is no subtree to return. YAFL

717c ⟨`SubTree` *as redefined by* `InterNonTerminalExpression` 717c⟩≡
```
REDEFINE METHOD SubTree: ARRAY OF InterComponent;
  BEGIN
  END SubTree;
```

At first sight, the following chunk might look totally useless and redundant. The task it performs is basically the same as the one performed by the `Class` method (see above).

However, in practice, this second version was made necessary by the fact that we need the ability to ensure that one refers to a class from the current specification (See method *DuraSpecification.SyntacticSpecification.FindClass* on page 298). In some obscure cases, when one attempts to generate a parser based on intermediate files that have been made obsolete by the deletion of one or more classes, but which have not been regenerated accordingly, the parser might produce corrupted parser. The additional checks that have been put in the `DuraTag` method ensure that such a faulty situation cannot occur without at least being reported.

718a ⟨`InterNonTerminalExpression` *semantic analysis* 718a⟩≡

```
METHOD DuraTag(Spec: SyntacticSpecification);
  VAR
    Words: ARRAY OF ARRAY OF CHAR;
  BEGIN
  Words ← String.BreakInWords(QualId,
                              SkipSeparator ← ‘‘.”,
                                 KeepSeparator ← Λ);
  TheClass ← Spec.FindClass (Space.StoreString (Words_0),
                             Space.StoreString (Words_1));
  IF TheClass = Λ THEN
    StdErr.WriteLine (‘‘Class ” + QualId +
                      ‘‘ not found within specification”);
    IF Spec.FindModule
           (Space.StoreString (Words_0)) = Λ THEN
      StdErr.WriteLine (‘‘Module not found”);
     ELSE
      StdErr.WriteLine (‘‘Module found”);
      END;
    ASSERT FALSE;
    END;
  END DuraTag;
```

`InterOptionalExpression` denotes a grammatical expression that refers to an optional operand, as in `Comma ?`.

718b ⟨`InterOptionalExpression` *class definition* 718b⟩≡

```
CLASS InterOptionalExpression;
  INHERITS InterExpression;

  METHOD SetOperand(Operand: InterExpression);
  METHOD Operand: InterExpression;
END InterOptionalExpression;
```

Well, there is very little worth mentioning in this class.

719 ⟨InterOptionalExpression *class implementation* 719⟩≡

```
CLASS InterOptionalExpression;
  INHERITS InterExpression;

  VAR
    TheOperand: InterExpression;

  METHOD SetOperand(Operand: InterExpression);
    BEGIN
    TheOperand ← Operand;
    END SetOperand;

  METHOD Operand: InterExpression;
    BEGIN
    RESULT ← TheOperand;
    END Operand;

  ⟨Parsing and unparsing an InterOptionalExpression 720a⟩

  REDEFINE METHOD SubTree: ARRAY OF InterComponent;
    BEGIN
    RESULT.CREATE (1);
    RESULT_0 ← Operand;
    END SubTree;

  ⟨BuildFlatReference, as redefined by InterOptionalExpression 729⟩

END InterOptionalExpression;
```

The unparsed representation of an `InterOptionalExpression` is made of a list where the first atom indicates that we are actually dealing with an optional expression while the second element of the list is a `Compound` (See page 801) that denotes the operand.

Parsing this operand is performed by the `ParseExpression` method (See page 759).

720a ⟨*Parsing and unparsing an* `InterOptionalExpression` 720a⟩≡
```
REDEFINE METHOD Parse(Comp: Compound);
  BEGIN
  ASSERT Comp.Size ≥ 2;
  ASSERT Comp_0.AtomCode = InteLex.Optional;
  TheOperand ← IntermediateParser.
                 ParseExpression(Comp_1;
  END Parse;

REDEFINE METHOD UnParseValue: Compound;
  BEGIN
  RESULT.CREATE;
  RESULT.AppendAtom (InteLex.Optional);
  RESULT.Append (Operand.UnParse);
  END UnParseValue;
```

`InterRepetitionExpression` denotes a repetition grammatical expression, that repeats an operand expression a variable number of times. It requires three pieces of information:

- The operand to repeat
- A flag indicating whether at least one repetition is required, or whether zero repetition is acceptable as well (`AtLeastOnce`).
- The separator expression, as in `Statement + BY SemiColon` where the `SemiColon` must separate the repeated instances of `Statement`.

720b ⟨`InterRepetitionExpression` *class definition* 720b⟩≡
```
CLASS InterRepetitionExpression;
  INHERITS InterExpression;

  METHOD AtLeastOnce: BOOLEAN;
  METHOD SetAtLeastOnce(Value: BOOLEAN);

  METHOD SetOperand(Operand: InterExpression);
  METHOD Operand: InterExpression;

  METHOD Separator: InterExpression;
  METHOD SetSeparator (Separator: InterExpression);

END InterRepetitionExpression;
```

721 ⟨InterRepetitionExpression *class implementation* 721⟩≡

```
CLASS InterRepetitionExpression;
  INHERITS InterExpression;

  VAR
    TheAtLeastOnce: BOOLEAN;
    TheOperand,
    TheSeparator: InterExpression;

  ⟨Setting and getting the attributes 722⟩
  ⟨Parsing and unparsing an InterRepetitionExpression instance 723a⟩

  REDEFINE METHOD SubTree: ARRAY OF InterComponent;
    BEGIN
    RESULT.CREATE (2);
    RESULT_0 ← Operand;
    RESULT_1 ← Separator;
    END SubTree;

  ⟨BuildFlatReference, as redefined by InterRepetitionExpression 732a⟩

END InterRepetitionExpression;
```

722 ⟨*Setting and getting the attributes* 722⟩≡

```
METHOD AtLeastOnce: BOOLEAN;
  BEGIN
  RESULT ← TheAtLeastOnce;
  END AtLeastOnce;

METHOD SetAtLeastOnce(Value: BOOLEAN);
  BEGIN
  TheAtLeastOnce ← Value;
  END SetAtLeastOnce;

METHOD SetOperand(Operand: InterExpression);
  BEGIN
  TheOperand ← Operand;
  END SetOperand;

METHOD Operand: InterExpression;
  BEGIN
  RESULT ← TheOperand;
  END Operand;

METHOD Separator: InterExpression;
  BEGIN
  RESULT ← TheSeparator;
  END Separator;

METHOD SetSeparator (Separator: InterExpression);
  BEGIN
  TheSeparator ← Separator;
  END SetSeparator;
```

723a ⟨*Parsing and unparsing an* InterRepetitionExpression *instance* 723a⟩≡
```
REDEFINE METHOD Parse(Comp: Compound);
  BEGIN
  CASE Comp0.AtomCode OF
    InteLex.Repeat1:
      SetAtLeastOnce (TRUE);
      END;
    InteLex.Repeat0:
      SetAtLeastOnce (FALSE);
      END;
    END;
  SetOperand(IntermediateParser.
                ParseExpression(Comp1;
  IF Comp.Size > 2 THEN
    SetSeparator(IntermediateParser.
                ParseExpression(Comp2;
    END;
  END Parse;
```

723b ⟨*Parsing and unparsing an* InterRepetitionExpression *instance* 723a⟩+≡
```
REDEFINE METHOD UnParseValue: Compound;
  BEGIN
  RESULT.CREATE;
  IF AtLeastOnce THEN
    RESULT.AppendAtom (InteLex.Repeat1);
   ELSE
    RESULT.AppendAtom (InteLex.Repeat0);
    END;
  RESULT.Append (Operand.UnParse);
  IF Separator ≠ Λ THEN
    RESULT.Append (Separator.UnParse);
    END;
  END UnParseValue;
```

InterCaseExpression denotes a grammatical expression that represents a disjunction with an explicit tag that indicates which of the alternatives of this disjunction has been taken into account (See page 261).

723c ⟨InterCaseExpression *class definition* 723c⟩≡
```
CLASS InterCaseExpression;
  INHERITS InterExpression;

  METHOD SetDiscriminantAttribute (Attr: InterAttribute);
  METHOD AppendAlt (Alt: InterCaseAlt);
END InterCaseExpression;
```

The alternatives of the disjunction are denoted by the `InterCaseAlt` class (See page 726). Each of these alternatives includes an integer value that uniquely identifies the alternative within the `InterCaseExpression`. An attribute (referred to by `TheDiscr`) will hold the integer value attached to the alternative of the disjunction that will have been used.

724 ⟨`InterCaseExpression` *class implementation* 724⟩≡

```
CLASS InterCaseExpression;
  INHERITS InterExpression;

  VAR
    TheDiscr: InterAttribute;
    TheAlts: ArrayList(InterCaseAlt);

  ⟨Setting the attributes in InterCaseExpression 725a⟩
  ⟨Parsing and unparsing an InterCaseExpression 725b⟩
  ⟨Building a flat reference to a InterCaseExpression 734a⟩

  REDEFINE METHOD SubTree: ARRAY OF InterComponent;
    BEGIN
    IF TheAlts = Λ THEN
      RESULT.CREATE (1);
     ELSE
      RESULT.CREATE (TheAlts.Size + 1);
      FOR i ∈ 0 .. TheAlts.Size - 1 DO
        RESULT_{i+1} ← TheAlts_i;
        END;
      END;
    RESULT_0 ← TheDiscr;
    END SubTree;

  ⟨RelatedAttributes as redefined by InterCaseExpression 726a⟩

END InterCaseExpression;
```

725a ⟨*Setting the attributes in* `InterCaseExpression` 725a⟩≡
```
METHOD SetDiscriminantAttribute (Attr: InterAttribute);
  BEGIN
  TheDiscr ← Attr;
  END SetDiscriminantAttribute;

METHOD AppendAlt (Alt: InterCaseAlt);
  BEGIN
  IF TheAlts = Λ THEN
    TheAlts.CREATE;
    END;
  TheAlts.Append (Alt);
  END AppendAlt;
```

The list representation of an `InterCaseExpression` is made of an identifying atom, a reference to the discriminant attribute, followed by the alternatives.

725b ⟨*Parsing and unparsing an* `InterCaseExpression` 725b⟩≡
```
REDEFINE METHOD Parse (Comp: Compound);
  VAR
    NewAlt: InterCaseAlt;
  BEGIN
  ASSERT Comp₀.CastToAtom.Code = InteLex.Case;
  TheDiscr ← IntermediateParser.ParseAttribute
                        (Comp₁;
  FOR i ∈ 2 .. Comp.Size - 1 DO
    NewAlt.CREATE;
    NewAlt.Parse (Compᵢ;
    AppendAlt (NewAlt);
    END;
  END Parse;
```

After parsing an `InterCaseExpression`, unparsing it is rather predictable...

725c ⟨*Parsing and unparsing an* `InterCaseExpression` 725b⟩+≡
```
REDEFINE METHOD UnParseValue: Compound;
  BEGIN
  RESULT.CREATE;
  RESULT.AppendAtom (InteLex.Case);
  RESULT.Append (TheDiscr.UnParse);
  FOR a ∈ TheAlts DO
    RESULT.Append (a.UnParse);
    END;
  END UnParseValue;
```

`InterCaseExpression` redefines `RelatedAttributes` (See page 705) to add the discriminant attribute of the CASE expression to the list of the attributes possibly impacted by this grammatical expression.

726a ⟨`RelatedAttributes` *as redefined by* `InterCaseExpression` 726a⟩≡

```
REDEFINE METHOD RelatedAttributes: ARRAY OF InterAttribute;
  BEGIN
  RESULT.CREATE (1);
  RESULT[0] ← TheDiscr;
  RESULT ← RESULT + BASE;
  END RelatedAttributes;
```

`InterCaseAlt` does not inherit from `InterExpression`. It is not a stand-alone grammatical expression. It is used within `InterCaseExpression` to denote the alternatives of the CASE expression.

726b ⟨`InterCaseAlt` *class definition* 726b⟩≡

```
CLASS InterCaseAlt;
  INHERITS InterComponent;

  METHOD SetExpression (Expr: InterExpression);
  METHOD SetDiscriminantValue (Value: INTEGER);
END InterCaseAlt;
```

Each of these alternatives holds a grammatical expression that will be used in the disjunction; and a discriminant integer value that will be fetched into the enclosing CASE expression's discriminant attribute if the alternative is used.

727 ⟨InterCaseAlt *class implementation* 727⟩≡

```
CLASS InterCaseAlt;
  INHERITS InterComponent;

  VAR
    TheExpr: InterExpression;
    TheDiscriminantValue: INTEGER;

  METHOD SetExpression (Expr: InterExpression);
    BEGIN
    TheExpr ← Expr;
    END SetExpression;

  METHOD SetDiscriminantValue (Value: INTEGER);
    BEGIN
    TheDiscriminantValue ← Value;
    END SetDiscriminantValue;

  ⟨Parsing and unparsing an InterCaseAlt 728a⟩

  REDEFINE METHOD SubTree: ARRAY OF InterComponent;
    BEGIN
    RESULT.CREATE (1);
    RESULT0 ← TheExpr;
    END SubTree;

  ⟨BuildFlatReference, as defined by InterCaseAlt 734b⟩

END InterCaseAlt;
```

A case alternative in the list form (that is, as a `Compound` (See page 801)) is made of an element that identifies the construct, followed by the attached discriminant value and grammatical expression. This expression is turned into a `Compound` by calling `UnParse` (See page 685), and turned back into the intermediate representation by calling `ParseExpression`, as defined by `IntermediateParser` (See page 759).

728a ⟨*Parsing and unparsing an* `InterCaseAlt` 728a⟩≡

```
REDEFINE METHOD ComputeUnParse: Compound;
  BEGIN
  RESULT.CREATE;
  RESULT.AppendAtom (InteLex.CaseAlt);
  RESULT.AppendInteger (TheDiscriminantValue);
  RESULT.Append (TheExpr.UnParse);
  END ComputeUnParse;

METHOD Parse (Comp: Compound);
  BEGIN
  ASSERT Comp.Size = 3;
  ASSERT Comp_0.CastToAtom.Code = InteLex.CaseAlt;
  SetDiscriminantValue (Comp_1;
  SetExpression (IntermediateParser.
                 ParseExpression(Comp_2;
  END Parse;
```

7.20.3.3 Transforming the intermediate representation in the flattened representation

The intermediate representation is made of *grammatical expressions,* combined by operators (repetitions, disjunction, etc.), while the parser actually gets generated from the *flattened representation* described extensively in the `DuraFlat` (See page 474) and `DuraFlSpec` (See page 562) modules.

The transformation of the intermediate representation into the flattened representation is performed by the `BuildFlatReference` method, which is meant to be redefined by all `InterExpression`-derived classes. It returns a `FlatRuleComponent` (See page 540), which is the elementay component of a flattened rule. In most cases, `BuildFlatReference` will build a `FlatRule` (See page 480) and return a reference to it. As a consequence, the flattened representation will include many superfluous intermediate rules that will be removed by the various optimization possibilities (See method *DuraFlSpec.FlatSpecification.Normalize* on page 568).

728b ⟨*Deferred methods to be redefined by* `InterExpression` *derived classes* 704b⟩+≡

```
DEFERRED METHOD BuildFlatReference: FlatRuleComponent;
```

Using literate programming, one can emphasize how some of YAFL's features have been used. For instance, the various redefinitions of `BuildFlatReference` listed here do not necessarily use `FlatRuleComponent` as return type, but, depending on the case, `FlatTerminal` or `FlatNonTerminal` that are derived from `FlatRuleComponent`. With a bit of thought, one can be be quite confident this does not lead to any dynamic typing problem. Redefining a method in such a way that it returns a subtype of the type returned by the original method is safe, even in cases where by virtue of polymorphism, the redefinition is called dynamically instead of the original version. It is an entirely different story if one considers redefining a method as taking a different set of parameters than the original method it redefines.

When flattening a grammatical expression such as `ExprA ?`, one must return a reference to a grammar rule $Synth$ defined by:

$$\begin{array}{lll} Synth & ::= & Rule_A \mid \\ & & \epsilon \end{array}$$

if $Rule_A$ is a reference to the rule obtained by flattening `ExprA`.

729 ⟨`BuildFlatReference`, *as redefined by* `InterOptionalExpression` 729⟩≡

```
REDEFINE METHOD BuildFlatReference: FlatNonTerminal;
  VAR
    FRule: FlatRule;
    FRHS: FlatRHS;
  BEGIN
  FRule.CREATE;
  FRHS.CREATE;
  FRHS.Append (Operand.FlatReference);
  FRule.Append(FRHS);
  FRHS.CREATE;
  FRule.Append(FRHS);
  RESULT ← FRule.FlatReference;
  END BuildFlatReference;
```

730a ⟨BuildFlatReference, *as redefined by* InterAggregateExpression 730a⟩≡

```
REDEFINE METHOD BuildFlatReference: FlatNonTerminal;

  VAR
    FRule: FlatRule;
    FRHS: FlatRHS;
  BEGIN
  FRule.CREATE;
  CASE Op OF
    DisjunctionOp:
      ⟨Handle disjunction 730b⟩
      END;
    ConcatOp:
      ⟨Handle concatenation 731a⟩
      END;
    END;
  RESULT ← FRule.FlatReference;
  END BuildFlatReference;
```

When flattening a grammatical expression such as `ExprA OR ExprB OR ExprC`, one must return a reference to a grammar rule *Synth* defined by:

$$\begin{array}{lll} Synth & ::= & Rule_A \mid \\ & & Rule_B \mid \\ & & Rule_C \end{array}$$

if $Rule_A$ (resp., $Rule_B$, $Rule_C$) is a reference to the rule obtained by flattening `ExprA` (resp., `ExprB`, `ExprC`).

730b ⟨*Handle disjunction* 730b⟩≡

```
FOR a ∈ Args DO
  FRHS.CREATE;
  FRHS.Append (a.FlatReference);
  FRule.Append (FRHS);
  END;
```

When flattening a grammatical expression such as `ExprA ExprB ExprC` (with implicit concatenation), one must return a reference to a grammar rule *Synth* defined by:

$$Synth \ ::= \ Rule_A \ Rule_B \ Rule_C$$

if $Rule_A$ (resp., $Rule_B$, $Rule_C$) is a reference to the rule obtained by flattening `ExprA` (resp., `ExprB`, `ExprC`).

731a ⟨*Handle concatenation* 731a⟩≡

```
ASSERT Op = ConcatOp;
FRHS.CREATE;
FOR a ∈ Args DO
  FRHS.Append (a.FlatReference);
  END;
FRule.Append (FRHS);
```

731b ⟨`BuildFlatReference` *as redefined by* `InterTerminalExpression` 731b⟩≡

```
REDEFINE METHOD BuildFlatReference: FlatTerminal;
  BEGIN
  RESULT.CREATE (TheLexCode);
  END BuildFlatReference;
```

731c ⟨`BuildFlatReference` *as redefined by* `InterTerminalSetExpression` 731c⟩≡

```
REDEFINE METHOD BuildFlatReference: FlatTerminalSet;
  BEGIN
  RESULT.CREATE(TheSet);
  END BuildFlatReference;
```

731d ⟨`BuildFlatReference` *as redefined by* `InterNonTerminalExpression` 731d⟩≡

```
REDEFINE METHOD BuildFlatReference: FlatNonTerminal;
  BEGIN
  IF Class ≠ Λ THEN
    RESULT ← Class.FlatReference;
    END;
  END BuildFlatReference;
```

When flattening a grammatical expression such as `ExprA + BY ExprC` one must return a reference to a grammar rule *Synth* defined by:

$$\begin{aligned} Synth \; ::= \; & Synth\ Rule_C\ Rule_A \mid \\ & Rule_A \end{aligned}$$

if $Rule_A$ (resp. $Rule_C$) is a reference to the rule obtained by flattening `ExprA` (resp. `ExprC`). Extrapolating this to the simpler case where no separator `ExprC` is specified is straightforward.

732a ⟨`BuildFlatReference`, *as redefined by* `InterRepetitionExpression` 732a⟩≡

```
REDEFINE METHOD BuildFlatReference: FlatNonTerminal;
  VAR
    FRule, FinalRule: FlatRule;
    FRHS: FlatRHS;
    Concat: ConcatListReduction;
    OperandComponent: FlatRuleComponent;
  BEGIN
  FRule.CREATE;
  FRHS.CREATE;
  FRule.Append(FRHS);
  FRHS.Append (FRule.FlatReference);
  ⟨If a separator is defined, append a reference to it to FRHS 732b⟩
  OperandComponent ← Operand.FlatReference;
  FRHS.Append (OperandComponent);
  ⟨CREATE and attach the reduction 733a⟩
  FRHS.CREATE;
  FRule.Append(FRHS);
  FRHS.Append (OperandComponent);
  IF AtLeastOnce THEN
    RESULT ← FRule.FlatReference;
   ELSE
    ⟨Generate an additional rule that can produce FRule or nothing 733b⟩
    END;
  END BuildFlatReference;
```

732b ⟨*If a separator is defined, append a reference to it to* `FRHS` 732b⟩≡

```
IF Separator ≠ Λ THEN
  FRHS.Append (Separator.FlatReference);
  END;
```

It is not enough for us to claim that `ExprA +` should turn into something of the form:

$$\begin{aligned} Synth \ ::= \ & Synth\ Rule_A \ | \\ & Rule_A \end{aligned}$$

we must also ensure that when the first of these rules is reduced, the leading $Synth$ and the trailing $Rule_A$ are linked together, so that they can be fetched as a list. The problem remains of course exactly the same whether a separator expression is specified using `BY` or not. This is taken care of by the an instance of `ConcatListReduction` (See page 830) that we create and attach to the corresponding RHS. This `ConcatListReduction` will generate the code that will perform this link operation when reducing a $Synth$.

733a ⟨*CREATE and attach the reduction* 733a⟩≡

```
Concat.CREATE;
FRHS.SetReduction (Concat);
```

When flattening a grammatical expression such as `ExprA * BY ExprC` one must return a reference to a grammar rule $Synth$ defined by:

$$\begin{aligned} Synth \ ::= \ & Synth' \ | \\ & \epsilon \end{aligned}$$

where $Synth'$ is a reference to the flattened representation of `ExprA + BY ExprC`.

733b ⟨*Generate an additional rule that can produce* `FRule` *or nothing* 733b⟩≡

```
FinalRule.CREATE;
FRHS.CREATE;
FRHS.Append (FRule.FlatReference);
FinalRule.Append (FRHS);
FRHS.CREATE;
FinalRule.Append (FRHS);
RESULT ← FinalRule.FlatReference;
```

Building the flattened representation for an `InterCaseExpression` is awfully similar to what has been described above for normal disjunctions. It is the alternative's responsibility to maintain the discriminant attribute's information (See method *DuraIntermediateFormat.InterCaseAlt.BuildFlatReference* on this page).

734a ⟨*Building a flat reference to a* `InterCaseExpression` 734a⟩≡
```
REDEFINE METHOD BuildFlatReference: FlatRuleComponent;
  VAR
    FRule: FlatRule;
    FRHS: FlatRHS;
  BEGIN
  FRule.CREATE;
  FOR a ∈ TheAlts DO
    FRHS.CREATE;
    FRHS.Append (a.BuildFlatReference(TheDiscr));
    FRule.Append (FRHS);
    END;
  RESULT ← FRule.FlatReference;
  END BuildFlatReference;
```

This is not a *redefinition* of `BuildFlatReference` since `InterCaseAlt` does not inherit from `InterExpression`. It basically creates a new grammar rule with a ringle RHS (right hand side) and attaches a newly created `IndicatorReduction`(See page 835) to it, so that the information about which discriminant value must be fetched into which attribute in case of reduction is not lost.

734b ⟨`BuildFlatReference`, *as defined by* `InterCaseAlt` 734b⟩≡
```
METHOD BuildFlatReference(DiscrAttr: InterAttribute):
                                            FlatRuleComponent;
  VAR
    FRule: FlatRule;
    FRHS: FlatRHS;
    IReduc: IndicatorReduction;
  BEGIN
  FRule.CREATE;
  FRHS.CREATE;
  FRule.Append (FRHS);
  FRHS.Append (TheExpr.FlatReference);
  IReduc.CREATE (DiscrAttr, TheDiscriminantValue);
  FRHS.SetReduction (IReduc);
  RESULT ← FRule.FlatReference;
  END BuildFlatReference;
```

7.20.4 Managing attributes at the intermediate level

735a ⟨`InterAttribute` *class definition* 735a⟩≡

```
CLASS InterAttribute;
  INHERITS InterComponent;

  REDEFINE METHOD CREATE;

  ⟨Setting and consulting attributes trivially, definition 736b⟩

  METHOD IsEqual (Other: InterAttribute): BOOLEAN;

 ⟨Meta-attribute variable declaration generation, definition 738d⟩
 ⟨Meta-attribute variable initialization generation, definition 739b⟩
 ⟨Meta-attribute variable fetch generation, definition 740a⟩

END InterAttribute;
```

We maintain several pieces of information about attributes at the intermediate level:

- Name, made of module, class, and attribute name
- Unique numbering to allow the convenient generation of unique identifiers
- Type (See class *DuraIntermediateFormat.InterType* on page 686)

735b ⟨`InterAttribute` *class implementation* 735b⟩≡

```
CLASS InterAttribute;
  INHERITS InterComponent;

  ⟨Creation and numbering management 736a⟩
  ⟨Setting and consulting attributes trivially, implementation 737⟩
  ⟨Parsing and unparsing an InterAttribute 738a⟩
  ⟨Checking whether two InterAttribute's are the same 738c⟩

  REDEFINE METHOD SubTree: ARRAY OF InterComponent;
    BEGIN
    END SubTree;

  ⟨Meta-attribute variable declaration generation, implementation 739a⟩
  ⟨Meta-attribute variable initialization generation, implementation 739c⟩
  ⟨Meta-attribute variable fetch generation, implementation 740b⟩

END InterAttribute;
```

736a ⟨*Creation and numbering management* 736a⟩≡

```
VAR
  TheNr: INTEGER;

REDEFINE METHOD CREATE;
  VAR
    Counter: ONCE INTEGER;
  BEGIN
  Counter ← Counter + 1;
  TheNr ← Counter;
  BASE;
  END CREATE;
```

736b ⟨*Setting and consulting attributes trivially, definition* 736b⟩≡

```
METHOD SetNames (AttrName, ClassName, ModuleName: ARRAY OF CHAR);
METHOD FullName: ARRAY OF CHAR;
METHOD SetType (Type: InterType);
METHOD Type: InterType;
METHOD EnclosingClass: DuraClassDefinition;
```

737 ⟨*Setting and consulting attributes trivially, implementation* 737⟩≡

```
VAR
  TheAttrName,
  TheClassName,
  TheModuleName: ARRAY OF CHAR;
  TheType: InterType;
  TheEnclosingClass: DuraClassDefinition;

METHOD SetNames (AttrName,
                 ClassName,
                   ModuleName: ARRAY OF CHAR);
  BEGIN
  TheAttrName ← AttrName;
  TheClassName ← ClassName;
  TheModuleName ← ModuleName;
  END SetNames;

METHOD FullName: ARRAY OF CHAR;
  BEGIN
  RESULT ← TheModuleName + ‘‘.” + TheClassName + ‘‘.” +
            TheAttrName;
  END FullName;

METHOD SetType (Type: InterType);
  BEGIN
  TheType ← Type;
  END SetType;

METHOD Type: InterType;
  BEGIN
  RESULT ← TheType;
  END Type;

METHOD EnclosingClass: DuraClassDefinition;
  BEGIN
  IF TheEnclosingClass = Λ THEN
    TheEnclosingClass ←
         DuraCfg.GetClass (TheModuleName + ‘‘.” +
                           TheClassName);
    END;
  RESULT ← TheEnclosingClass;
  END EnclosingClass;
```

738a ⟨*Parsing and unparsing an* `InterAttribute` 738a⟩≡

```
REDEFINE METHOD ComputeUnParse: Compound;
   BEGIN
   RESULT.CREATE;
   RESULT.AppendAtom (InteLex.Attribute);
   RESULT.AppendAtomText (InteLex.Ident, TheModuleName);
   RESULT.AppendAtomText (InteLex.Ident, TheClassName);
   RESULT.AppendAtomText (InteLex.Ident, TheAttrName);
   RESULT.Append (Type.UnParse);
   END ComputeUnParse;
```

When parsing an attribute, we must store the strings related to the module, class and attribute name in the string space `Space` provided as part of the YAFL compiler to allow the comparison for string equality to be performed by reference rather than by value. This ability is used in the `IsEqual` method (see below).

738b ⟨*Parsing and unparsing an* `InterAttribute` 738a⟩+≡

```
METHOD Parse(Comp: Compound);
   BEGIN
   ASSERT Comp.Size = 5;
   ASSERT Comp_0.AtomCode = InteLex.Attribute;
   SetNames (Space.StoreString(Comp_3,
             Space.StoreString(Comp_2,
             Space.StoreString(Comp_1;
   SetType (IntermediateParser.ParseType (Comp_4;
   END Parse;
```

738c ⟨*Checking whether two* `InterAttribute`*'s are the same* 738c⟩≡

```
METHOD IsEqual (Other: InterAttribute): BOOLEAN;
   BEGIN
   RESULT ← (TheAttrName = Other.TheAttrName) ∧
            (TheClassName = Other.TheClassName) ∧
            (TheModuleName = Other.TheModuleName);
   END IsEqual;
```

`GenerateMetaAttribute` generates the name of the meta-attribute attached to `THIS`. This name is generated using the numbering system, and the `TheNr` attribute (See page 736). The meta-attribute is an object that allows us to bypass YAFL encapsulation mechanism, and fetch data into attributes directly without going through access methods. `DURA` produces one of such meta-attributes for each attribute which must be given a value when producing a parse tree.

`GenerateMetaAttributeDecl` generates a valid YAFL attribute declaration for the meta-attribute, using the `GenerateMetaAttributeType` method for the attached type (See page 686).

738d ⟨*Meta-attribute variable declaration generation, definition* 738d⟩≡

```
METHOD GenerateMetaAttribute (Output: DuraOutputStream);
METHOD GenerateMetaAttributeDecl (Output: DuraOutputStream);
```

739a ⟨*Meta-attribute variable declaration generation, implementation* 739a⟩≡
```
METHOD GenerateMetaAttribute (Output: DuraOutputStream);
  BEGIN
  Output.WriteString (‘‘A_” +
                IntConversions.IntToString (TheNr, 0));
  END GenerateMetaAttribute;

METHOD GenerateMetaAttributeDecl (Output: DuraOutputStream);
  BEGIN
  GenerateMetaAttribute (Output);
  Output.WriteString (‘‘: ”);
  TheType.GenerateMetaAttributeType (Output);
  Output.WriteLine (‘‘; -- Attribute name : ” + FullName);
  END GenerateMetaAttributeDecl;
```

739b ⟨*Meta-attribute variable initialization generation, definition* 739b⟩≡
```
METHOD GenerateMetaAttributeInit (Output: DuraOutputStream;
                                  Class: DuraClassDefinition);
```

`GenerateMetaAttributeInit` generates YAFL code to initialize the meta-attribute, by finding the attribute in the YAFL metaclass system, and then refining its type using a WHAT statement.

The actual search for the attribute it performed by the `FindAttribute` method as defined in the skeleton (See page 871).

739c ⟨*Meta-attribute variable initialization generation, implementation* 739c⟩≡
```
METHOD GenerateMetaAttributeInit (Output: DuraOutputStream;
                                  Class: DuraClassDefinition);
  BEGIN
  Output.WriteString (‘‘WHAT FindAttribute(”);
  Class.GenerateMetaClassVariable (Output);
  Output.WriteLine(‘‘, ’” + TheAttrName + ‘‘’) OF”);
  Output.IncIndent (2);
  Output.WriteString (‘‘IN ”);
  TheType.GenerateMetaAttributeType (Output);
  Output.WriteLine (‘‘:”);
  Output.IncIndent (2);
  GenerateMetaAttribute (Output);
  Output.WriteLine (‘‘ := TAG;”);
  Output.End;
  Output.End;
  END GenerateMetaAttributeInit;
```

The GenerateFetch method presented below generates the code that fetches parts of a non-terminal that has been recognized into the attribute that is denoted by THIS. Index denotes the index within an array TheArr where the LALR stack element that must be fetch can be found.

740a ⟨*Meta-attribute variable fetch generation, definition* 740a⟩≡

```
METHOD GenerateFetch (Output: DuraOutputStream;
                      Index: INTEGER);
```

Basically, we leave the responsibility to THIS's type to generate the proper code to fetch, except for the fact that the metaclass to which the meta-attribute attached to THIS is connected must refer to a valid instance. In other words, when fetching a *Condition* into an *IfStatement*, we must first make sure that the metaclass attached to *IfStatement*'s does actually refer to a valid *IfStatement* instance. Otherwise, fetching the *Condition* will result in a runtime error.

740b ⟨*Meta-attribute variable fetch generation, implementation* 740b⟩≡

```
METHOD GenerateFetch (Output: DuraOutputStream;
                      Index: INTEGER);
  BEGIN
  Output.WriteString (``IF ");
  EnclosingClass.GenerateMetaClassVariable(Output);
  Output.IncIndent (2);
  Output.WriteLine(``.Instance = VOID THEN");
  Output.WriteLine(``Warning ('Cannot fetch " + FullName + ``');");
  Output.Else;
  Type.GenerateFetch (Output, Index, THIS);
  Output.End;
  END GenerateFetch;
```

7.20.5 Representing THIS at the intermediate level

InterThis represents a very peculiar kind of so-called attribute which is used in conjunction with *assignments to THIS* (See page 271). It is not a true attribute, but it can be used as target for grammar expressions.

740c ⟨InterThis *class definition* 740c⟩≡

```
ONCE CLASS InterThis;
  INHERITS InterAttribute;
END InterThis;
```

741a ⟨`InterThis` *class implementation* 741a⟩≡

```
ONCE CLASS InterThis;
  INHERITS InterAttribute;

  REDEFINE METHOD FullName: ARRAY OF CHAR;
    BEGIN
    RESULT ← PredefItems.This.Id.Data;
    END FullName;

 REDEFINE METHOD ComputeUnParse: Atom;
   BEGIN
   RESULT.CREATE (Λ, InteLex.This);
   END ComputeUnParse;

  ⟨Code generation methods, as redefined by InterThis 741b⟩

END InterThis;
```

Since `InterThis` does not really denote an attribute, all the code generation methods are redefined as empty methods. Handling the special case of assignment to THIS will be managed by the code generation of reductions.

741b ⟨*Code generation methods, as redefined by* `InterThis` 741b⟩≡

```
REDEFINE METHOD GenerateMetaAttribute
                          (Output: DuraOutputStream);
  BEGIN
  END GenerateMetaAttribute;

REDEFINE METHOD GenerateMetaAttributeDecl
                          (Output: DuraOutputStream);
  BEGIN
  END GenerateMetaAttributeDecl;

REDEFINE METHOD GenerateMetaAttributeInit
                          (Output: DuraOutputStream;
                           Class: DuraClassDefinition);
  BEGIN
  END GenerateMetaAttributeInit;

REDEFINE METHOD GenerateFetch (Output: DuraOutputStream;
                                    Index: INTEGER);
  BEGIN
  END GenerateFetch;
```

7.20.6 Representing grammar rules at the intermediate level

Now that we've been through the representation of the most granular pieces of information involved in grammatical expressions, let's now investigate how we deal with higher-level issues, such as rules, clauses and classes.

Each non-terminal class is attached to a grammar clause (See class *DuraGrClause. GrammarClause* on page 666), and each of these clauses can contain one or more grammar rules (See the `InterRule` class below). By default, all the grammar rules within a clause inherit from its properties (recovery policy, snip level) but they can override them explicitly.

742 ⟨`InterRule` *class definition* 742⟩≡

```
CLASS InterRule;
  INHERITS InterComponent;

  REDEFINE METHOD CREATE(EnclosingClause: InterClause);
  METHOD EnclosingClause: InterClause;

  ⟨Associativity handling method definitions 743b⟩
  ⟨Precedence handling method definitions 745b⟩
  ⟨Snip level handling method definitions 744b⟩
  ⟨Recovery level handling method definitions 744e⟩

  ⟨Setting and consulting the attached expression, definition 745e⟩
  ⟨FlatReference method definition for InterRule 747a⟩


END InterRule;
```

743a ⟨InterRule *class implementation* 743a⟩≡

```
CLASS InterRule;
  INHERITS InterComponent;

  VAR
    ⟨InterRule Attributes 743c⟩
    TheEnclosingClause: InterClause;

  REDEFINE METHOD CREATE(EnclosingClause: InterClause);
    BEGIN
    BASE;
    TheEnclosingClause ← EnclosingClause;
    END CREATE;

  METHOD EnclosingClause: InterClause;
    BEGIN
    RESULT ← TheEnclosingClause;
    END EnclosingClause;

  ⟨Setting and consulting the attached expression, implementation 745g⟩
  ⟨Associativity handling method implementations 744a⟩
  ⟨Precedence handling method implementations 745d⟩
  ⟨Snip level handling method implementations 744d⟩
  ⟨Recovery level handling method implementations 745a⟩
  ⟨Parsing and unparsing an InterRule 746a⟩

  REDEFINE METHOD SubTree: ARRAY OF InterComponent;
    BEGIN
    RESULT.CREATE (1);
    RESULT_0 ← Expression;
    END SubTree;

  ⟨FlatReference method implementation for InterRule 747b⟩

END InterRule;
```

743b ⟨*Associativity handling method definitions* 743b⟩≡

```
CONST
  LeftAssoc =  GrammarRule.LeftAssoc;
  NoAssoc =    GrammarRule.NoAssoc;
  RightAssoc = GrammarRule.RightAssoc;

METHOD Associativity: INTEGER;
METHOD SetAssociativity(Value: INTEGER);
```

743c ⟨InterRule *Attributes* 743c⟩≡

```
TheAssociativity: INTEGER;
```

744a ⟨*Associativity handling method implementations* 744a⟩≡
```
METHOD Associativity: INTEGER;
  BEGIN
  RESULT ← TheAssociativity;
  END Associativity;

METHOD SetAssociativity(Value: INTEGER);
  BEGIN
  TheAssociativity ← Value;
  END SetAssociativity;
```

744b ⟨*Snip level handling method definitions* 744b⟩≡
```
CONST
  NoCut = GrammarRule.NoCut;
  DefaultCut = GrammarRule.DefaultCut;
  FullCut = GrammarRule.FullCut;

METHOD CutLevel: INTEGER;
METHOD SetCutLevel(Value: INTEGER);
```

744c ⟨InterRule *Attributes* 743c⟩+≡
```
TheCutLevel : INTEGER;
```

744d ⟨*Snip level handling method implementations* 744d⟩≡
```
METHOD CutLevel: INTEGER;
  BEGIN
  RESULT ← TheCutLevel;
  END CutLevel;

METHOD SetCutLevel(Value: INTEGER);
  BEGIN
  TheCutLevel ← Value;
  END SetCutLevel;
```

744e ⟨*Recovery level handling method definitions* 744e⟩≡
```
CONST
  NoRecovery = GrammarRule.NoRecovery;
  DefaultRecovery = GrammarRule.DefaultRecovery;
  AutomaticRecovery = GrammarRule.AutomaticRecovery;

METHOD RecoveryLevel: INTEGER;
METHOD SetRecoveryLevel(Value: INTEGER);
```

744f ⟨InterRule *Attributes* 743c⟩+≡
```
TheRecoveryLevel : INTEGER;
```

745a ⟨*Recovery level handling method implementations* 745a⟩≡
```
METHOD RecoveryLevel: INTEGER;
  BEGIN
  RESULT ← TheRecoveryLevel;
  END RecoveryLevel;

METHOD SetRecoveryLevel(Value: INTEGER);
  BEGIN
  TheRecoveryLevel ← Value;
  END SetRecoveryLevel;
```

745b ⟨*Precedence handling method definitions* 745b⟩≡
```
METHOD Precedence: INTEGER;
METHOD SetPrecedence (Precedence: INTEGER);
```

745c ⟨`InterRule` *Attributes* 743c⟩+≡
```
ThePrecedence : INTEGER;
```

745d ⟨*Precedence handling method implementations* 745d⟩≡
```
METHOD Precedence: INTEGER;
  BEGIN
  RESULT ← ThePrecedence;
  END Precedence;

METHOD SetPrecedence (Precedence: INTEGER);
  BEGIN
  ThePrecedence ← Precedence;
  END SetPrecedence;
```

The following chunks manage the `InterExpression` (See page 703) attached to the grammar rule.

745e ⟨*Setting and consulting the attached expression, definition* 745e⟩≡
```
METHOD Expression: InterExpression;
METHOD SetExpression (Expression: InterExpression);
```

745f ⟨`InterRule` *Attributes* 743c⟩+≡
```
TheExpression: InterExpression;
```

745g ⟨*Setting and consulting the attached expression, implementation* 745g⟩≡
```
METHOD Expression: InterExpression;
  BEGIN
  RESULT ← TheExpression;
  END Expression;

METHOD SetExpression (Expression: InterExpression);
  BEGIN
  TheExpression ← Expression;
  END SetExpression;
```

An `InterRule`, when represented as a list, is made of a leading identifier, followed by the associativity, the precedence, the cut level, the recovery level, and the attached grammatical expression.

746a ⟨*Parsing and unparsing an* `InterRule` 746a⟩≡

```
REDEFINE METHOD ComputeUnParse: Compound;
  BEGIN
  RESULT.CREATE;
  RESULT.AppendAtom (InteLex.GrammarRule);
  RESULT.AppendInteger (Associativity);
  RESULT.AppendInteger (Precedence);
  RESULT.AppendInteger (CutLevel);
  RESULT.AppendInteger (RecoveryLevel);
  RESULT.Append (TheExpression.UnParse);
  END ComputeUnParse;
```

Based on the `ComputeUnParse` method described above, `Parse` is pretty straightforward . . .

746b ⟨*Parsing and unparsing an* `InterRule` 746a⟩+≡

```
METHOD Parse (Comp: Compound);
  BEGIN
  ASSERT Comp.Size = 6;
  ASSERT Comp0.AtomCode = InteLex.GrammarRule;
  ASSERT ∀ i ∈ 1 .. 4 • Compi.AtomCode = InteLex.Integer;
  TheAssociativity ← Comp1.CastToAtom.IntegerValue;
  ThePrecedence ← Comp2.CastToAtom.IntegerValue;
  TheCutLevel ← Comp3.CastToAtom.IntegerValue;
  TheRecoveryLevel ← Comp4.CastToAtom.IntegerValue;
  TheExpression ← IntermediateParser.
                    ParseExpression(Comp5;
  END Parse;
```

`FlatReference`, when applied to a `InterRule` instance, returns a reference (See class *DuraFlat.FlatNonTerminal* on page 554) to a grammar rule made of a single flattened right hand side. This RHS matches the rule's grammar expression. The various subparts are replaced by synthetic rule references if necessary.

For instance, consider the following grammar rule, that uses extended operators:

```
GRAMMAR
  IF Expression THEN StatementList (ELSE StatementList)? END;
```

Calling `FlatReference` on this rule will return a reference to something similar to $Rule_{117}$ in

$$\begin{array}{lll} Rule_{117} & ::= & Rule_{118} \\ Rule_{118} & ::= & \mathit{IF}\ Rule_{64}\ \mathit{THEN}\ Rule_{13}\ Rule_{119}\ \mathit{END} \\ Rule_{119} & ::= & \mathit{ELSE}\ Rule_{13}\ | \\ & & \epsilon \end{array}$$

assuming that $Rule_{64}$ has been entered as referencing `Expression` and $Rule_{13}$ as referencing `StatementList`.

The last production expresses the fact that $Rule_{119}$ can be reduced to an empty string.

The phony reduction $Rule_{117}\ ::=\ Rule_{118}$ will be optimized by a later pass on the flattened representation of the grammar (See method *DuraFlSpec.FlatSpecification. Normalize* on page 568).

747a ⟨`FlatReference` *method definition for* `InterRule` 747a⟩≡

```
METHOD FlatReference: FlatNonTerminal;
```

747b ⟨`FlatReference` *method implementation for* `InterRule` 747b⟩≡

```
VAR
  TheFlatReference: FlatNonTerminal;

METHOD FlatReference: FlatNonTerminal;
VAR
  FetchNTReduct: FetchNonTerminalReduction;
  FRule: FlatRule;
  RHS: FlatRHS;
BEGIN
IF TheFlatReference = Λ THEN
  RHS ← Expression.BuildFlatRHS;
```

We now create and attach a `FetchNonTerminalReduction` (See page 825) to the newly created RHS. This reduction object will ensure that when the reduction occurs, an instance of the attached class is created before the various attributes are fetched from the reduction.

747c ⟨`FlatReference` *method implementation for* `InterRule` 747b⟩+≡

```
FetchNTReduct.CREATE (THIS);
RHS.SetReduction (FetchNTReduct);
```

We now transfer the cut and recovery level to this newly created RHS . . .

748a ⟨FlatReference *method implementation for* InterRule 747b⟩+≡

```
    RHS.SetCutLevel (CutLevel);
    RHS.SetRecoveryLevel (RecoveryLevel);
    FRule.CREATE;
    FRule.Append (RHS);
    TheFlatReference.CREATE;
    TheFlatReference.SetRule (FRule);
    END;
  RESULT ← TheFlatReference;
  END FlatReference;
```

After seeing *rules*, that is, grammar expressions with extended operators and parsing strategy information (associativity, etc...), *clauses* are nothing more than a non-empty set of rules that are attached to a given non-terminal class, with an implicit disjunction.

Quite naturally, InterClauses are attached to classes, and contain a list of InterRules.

748b ⟨InterClause *class definition* 748b⟩≡

```
  CLASS InterClause;
    INHERITS InterComponent;

    REDEFINE METHOD CREATE;
    METHOD Append (IRule: InterRule);
    METHOD RuleList: ArrayList(InterRule);

    METHOD ClassName: ARRAY OF CHAR;
    METHOD SetClass (Class: DuraClassDefinition);
    METHOD Class: DuraClassDefinition;
    METHOD DuraTag(Spec: SyntacticSpecification);

    ⟨Snip handling, definition 753a⟩
    ⟨Recovery handling, definition 753c⟩

    METHOD ListTypes: ArrayList(InterListType);

  END InterClause;
```

749 ⟨InterClause *class implementation* 749⟩≡

```
CLASS InterClause;
  INHERITS InterComponent;

  VAR
    TheClass: DuraClassDefinition;
    TheClassName: ARRAY OF CHAR;
    TheRuleList: ArrayList(InterRule);

  ⟨Parsing and unparsing InterClause 750a⟩

  REDEFINE METHOD CREATE;
    BEGIN
    BASE;
    TheRuleList.CREATE;
    END CREATE;

  ⟨Managing the rule list 751b⟩
  ⟨Attaching and consulting the class name 751c⟩
  ⟨Attaching and consulting the class 752a⟩

  REDEFINE METHOD SubTree: ARRAY OF InterComponent;
    BEGIN
    RESULT.CREATE (TheRuleList.Size);
    FOR i ∈ 0 .. RESULT.SIZE - 1 DO
      RESULT_i ← TheRuleList_i;
      END;
    END SubTree;

  ⟨ListTypes method implementation 752b⟩
  ⟨Snip handling, implementation 753b⟩
  ⟨Recovery handling, implementation 753d⟩
  ⟨Semantic analysis on InterClause 754a⟩

END InterClause;
```

When represented as a lisp-like list, an `InterClause` is made of an identifier, followed by the name of the class it is included in, the cut level, the recovery level, followed by the flattened representation of the rules included in the clause (See method *DuraIntermediateFormat.InterRule.ComputeUnParse* on page 746).

750a ⟨*Parsing and unparsing* `InterClause` 750a⟩≡

```
  REDEFINE METHOD ComputeUnParse: Compound;
    VAR
      Temp: Compound;
    BEGIN
    RESULT.CREATE;
    RESULT.AppendAtom (InteLex.GrammarClause);
    RESULT.AppendAtomText (InteLex.Ident, ClassName);
    RESULT.AppendInteger (CutLevel);
    RESULT.AppendInteger (RecoveryLevel);
    Temp.CREATE;
    FOR Rule ∈ TheRuleList DO
      Temp.Append (Rule.UnParse);
      END;
    RESULT.Append (Temp);
    END ComputeUnParse;
```

Given what we have seen above for unparsing it, parsing the lisp-like representation is straightforward . . .

750b ⟨*Parsing and unparsing* `InterClause` 750a⟩+≡

```
  METHOD Parse (Comp: Compound);
    VAR
      IRule: InterRule;
    BEGIN
    ⟨Check for the validity of the Compound to parse 751a⟩
    SetClassName (Comp_1);
    SetCutLevel(Comp_2);
    SetRecoveryLevel(Comp_3);
    FOR e ∈ Comp_4.CastToCompound.Elements DO
      IRule.CREATE(THIS);
      IRule.Parse (e.CastToCompound);
      Append (IRule);
      END;
    END Parse;
```

751a ⟨*Check for the validity of the* `Compound` *to parse* 751a⟩≡
```
DEBUG
  IF (Comp0.AtomCode ≠ InteLex.GrammarClause) ∨
      (Comp1.AtomCode ≠ InteLex.Ident) THEN
    StdErr.WriteLine ("Err in ParseInterClause: ");
    Comp.Dump (StdErr, 4);
    ASSERT FALSE;
    END;
  END;
```

751b ⟨*Managing the rule list* 751b⟩≡
```
METHOD RuleList: ArrayList(InterRule);
  BEGIN
  RESULT ← TheRuleList;
  END RuleList;

METHOD Append (IRule: InterRule);
  BEGIN
  TheRuleList.Append (IRule);
  END Append;
```

When setting the class name, the character string is normalized by using the `Space` string space, provided as part of the YAFL compiler. This normalization process enables us to use reference equality for strings instead of value-based equality. YAFL

751c ⟨*Attaching and consulting the class name* 751c⟩≡
```
METHOD SetClassName (ClassName: ARRAY OF CHAR);
  BEGIN
  TheClassName ← Space.StoreString(ClassName);
  END SetClassName;

METHOD ClassName: ARRAY OF CHAR;
  BEGIN
  IF Class = Λ THEN
    RESULT ← TheClassName;
   ELSE
    RESULT ← Class.Id.Data;
    END;
  END ClassName;
```

`TheClass` and `TheClassName` are redundant, in the sense that `TheClass` includes a name. Hence, to avoid inconsistencies if any, whenever we set `TheClass`, we rest `TheClassName`.

752a ⟨*Attaching and consulting the class* 752a⟩≡
```
METHOD SetClass (Class: DuraClassDefinition);
  BEGIN
  TheClass ← Class;
  TheClassName ← Λ;
  END SetClass;

METHOD Class: DuraClassDefinition;
  BEGIN
  RESULT ← TheClass;
  END Class;
```

`ListTypes` builds and return a list that contains all the references to list types (incarnated by `InterListType` instances) in the grammar clause referred to by `THIS`.

In practice, we are only interested in attributes of such list types. Hence, we check for all `InterExpressions` (See page 703) using the `GrabSubNodes` method (See page 685), then only consider those that have a target (that is, that are assigned to an attribute), and finally, check whether the target's type is a list type.

752b ⟨`ListTypes` *method implementation* 752b⟩≡
```
METHOD ListTypes: ArrayList(InterListType);
  VAR
    ExprL: ArrayList(InterExpression);
  BEGIN
  ExprL.CREATE;
  GrabSubNodes (ExprL);
  FOR Expr ∈ ExprL |
       (Expr.Target ≠ Λ) ∧ (Expr.Target ≠ InterThis) DO
    WHAT Expr.Target.Type OF
      IN InterListType:
        IF RESULT = Λ THEN
          RESULT.CREATE;
          END;
        RESULT.Append (TAG);
        END;
     ELSE
      END;
    END;
  END ListTypes;
```

753a ⟨*Snip handling, definition* 753a⟩≡

```
CONST
  NoCut = DuraClassDefinition.NoCut;
  DefaultCut = DuraClassDefinition.DefaultCut;
  FullCut = DuraClassDefinition.FullCut;

METHOD CutLevel: INTEGER;
METHOD SetCutLevel (CutLevel: INTEGER);
```

753b ⟨*Snip handling, implementation* 753b⟩≡

```
VAR
  TheCutLevel: INTEGER;

METHOD CutLevel: INTEGER;
  BEGIN
  RESULT ← TheCutLevel;
  END CutLevel;

METHOD SetCutLevel (CutLevel: INTEGER);
  BEGIN
  TheCutLevel ← CutLevel;
  END SetCutLevel;
```

753c ⟨*Recovery handling, definition* 753c⟩≡

```
CONST
  NoRecovery =         DuraClassDefinition.NoRecovery;
  DefaultRecovery =    DuraClassDefinition.DefaultRecovery;
  AutomaticRecovery = DuraClassDefinition.AutomaticRecovery;

METHOD SetRecoveryLevel (Level: INTEGER);
METHOD RecoveryLevel: INTEGER;
```

753d ⟨*Recovery handling, implementation* 753d⟩≡

```
VAR
  TheRecoveryLevel: INTEGER;

METHOD SetRecoveryLevel (Level: INTEGER);
  BEGIN
  TheRecoveryLevel ← Level;
  END SetRecoveryLevel;

METHOD RecoveryLevel: INTEGER;
  BEGIN
  RESULT ← TheRecoveryLevel;
  END RecoveryLevel;
```

Since `InterNonTerminalExpression` (See page 715) is the only class on which semantic analysis must be performed, we first gather all the instances of this class by using the `GrabSubNodes` method (See page 685) and then perform semantic analysis on them (See method *DuraIntermediateFormat.InterNonTerminalExpression.DuraTag* on page 717).

754a ⟨*Semantic analysis on* `InterClause` 754a⟩≡

```
METHOD DuraTag(Spec: SyntacticSpecification);
  VAR
    l: ArrayList(InterNonTerminalExpression);
  BEGIN
  l.CREATE;
  GrabSubNodes (l);
  FOR Exp ∈ l DO
    Exp.DuraTag(Spec);
    END;
  END DuraTag;
```

After having seen how to handle the grammatical information related to a single class in `InterClause`, we now explain how the next level is handled, that is, when the various grammatical clauses found in a module are grouped together.

754b ⟨`InterModule` *class definition* 754b⟩≡

```
CLASS InterModule;
  INHERITS InterComponent;

  REDEFINE METHOD CREATE;
  METHOD Append (IClause: InterClause);

  METHOD FindClause (ClassName: ARRAY OF CHAR): InterClause;
    PRE
      ClassName = Space.StoreString(ClassName);
    POST
      RESULT ≠ Λ ⇒ RESULT.ClassName = ClassName;

  METHOD ParseFile (FName: ARRAY OF CHAR): BOOLEAN;

  METHOD ModuleName: ARRAY OF CHAR;
  METHOD SetModuleName (ModuleName: ARRAY OF CHAR);

END InterModule;
```

Basically, an `InterModule` is just a list of `InterClause` (See page 748) implemented in `TheList` together with a module name in `TheModuleName`.

755a ⟨`InterModule` *class implementation* 755a⟩≡

```
CLASS InterModule;
  INHERITS InterComponent;

  VAR
    TheList: ArrayList(InterClause);
    TheModuleName: ARRAY OF CHAR;

  REDEFINE METHOD CREATE;
    BEGIN
    BASE;
    TheList.CREATE;
    END CREATE;

  ⟨Managing InterModule's structure 755b⟩
  ⟨Parsing and unparsing an InterModule 756a⟩
  ⟨Finding an InterClause within an InterModule 757a⟩

  REDEFINE METHOD SubTree: ARRAY OF InterComponent;
    BEGIN
    RESULT.CREATE (TheList.Size);
    FOR i ∈ 0 .. RESULT.SIZE - 1 DO
      RESULT_i ← TheList_i;
      END;
    END SubTree;

END InterModule;
```

755b ⟨*Managing* `InterModule`*'s structure* 755b⟩≡

```
METHOD ModuleName: ARRAY OF CHAR;
  BEGIN
  RESULT ← TheModuleName;
  END ModuleName;

METHOD SetModuleName (ModuleName: ARRAY OF CHAR);
  BEGIN
  TheModuleName ← Space.StoreString(ModuleName);
  END SetModuleName;

METHOD Append (IClause: InterClause);
  BEGIN
  TheList.Append (IClause);
  END Append;
```

When parsing an intermediate file back into the intermediate representation, we first read a compound from the disk file (See method *DuraLisp.Compound.Read* on page 804), read the leading module name, and build a clause for each subsequent element in the `Compound`.

756a ⟨*Parsing and unparsing an* `InterModule` 756a⟩≡

```
METHOD ParseFile (FName: ARRAY OF CHAR): BOOLEAN;
  VAR
    Comp: Compound;
    IClause: InterClause;
  BEGIN
  Comp.CREATE;
  RESULT ← Comp.Read (FName);
  IF RESULT THEN
    SetModuleName (Comp_1;
    FOR El ∈ Comp_2.CastToCompound.Elements DO
      IClause.CREATE;
      IClause.Parse(El.CastToCompound);
      Append(IClause);
      END;
    END;
  END ParseFile;
```

Unparsing matches the structure for parsing closely . . .

756b ⟨*Parsing and unparsing an* `InterModule` 756a⟩+≡

```
REDEFINE METHOD ComputeUnParse: Compound;
  VAR
    Temp: Compound;
  BEGIN
  RESULT.CREATE;
  RESULT.AppendAtom (InteLex.Module);
  RESULT.AppendAtomText (InteLex.Ident, ModuleName);
  Temp.CREATE;
  FOR Clause ∈ TheList DO
    Temp.Append (Clause.UnParse);
    END;
  RESULT.Append (Temp);
  END ComputeUnParse;
```

`FindClause` returns the grammar clause in `THIS` which name matches `ClassName`. By using normalization when setting an `InterClause`'s name (See method *DuraIntermediateFormat.InterClause.SetClassName* on page 751), we can use reference equality ("=") for string comparison, assuming of course that `ClassName` has been normalized as well.

757a ⟨*Finding an* `InterClause` *within an* `InterModule` 757a⟩≡
```
METHOD FindClause (ClassName: ARRAY OF CHAR): InterClause;
  BEGIN
  RESULT ← FIRST Cl ∈ TheList • ClassName = Cl.ClassName;
  END FindClause;
```

7.20.7 Parsing the intermediate representation

In an ideal world, we would like to be able to parse having objects parsing themselves. In practice, we sometimes need a *lookahead* to decide which kind of object we must create before letting it parse itself.

`IntermediateParser` provides the conveniency methods that read whatever has to be read before selecting the right class to create, and then let the newly created object parse itself.

757b ⟨*Once class that recognizes and parses some* `InterComponent`*s* 757b⟩≡
```
ONCE CLASS IntermediateParser;

  ⟨Recognizing a type described by an Atom or a Compound 757c⟩
  ⟨Recognizing an attribute described by an Atom or a Compound 759a⟩
  ⟨Recognizing an expression described by a Compound 759b⟩

END IntermediateParser;
```

757c ⟨*Recognizing a type described by an* `Atom` *or a* `Compound` 757c⟩≡
```
METHOD ParseType (Comp: Component): InterType;
  VAR
    TheType: InterNonTerminalType;
    LType: InterListType;
  BEGIN
  WHAT Comp OF
    IN Compound:
      ⟨Recognize and rebuild an InterList representation 758b⟩
      END;
    IN Atom:
      ⟨Recognize any other type 758a⟩
      END;
    END;
  END ParseType;
```

If the `Component` we parse is an `Atom`, it can be a predefined type, or a reference to a non-terminal type.

758a ⟨*Recognize any other type* 758a⟩≡
```
CASE TAG.Code OF
  InteLex.IntegerType:
    RESULT ← InterIntegerType;
    END;
  InteLex.StringType:
    RESULT ← InterStringType;
    END;
  InteLex.BooleanType:
    RESULT ← InterBooleanType;
    END;
  InteLex.QualId:
    TheType.CREATE;
    TheType.SetQualId (TAG.Text);
    RESULT ← TheType;
    END;
  END;
```

If the `Component` we parse is a `Compound`, it must represent a list type. The following chunk performs the analysis, making extensive use of the `GetClass` method (See page 375) to convert a string representation of a class (using the $Module.Class$ syntax) into a reference to the class.

758b ⟨*Recognize and rebuild an* `InterList` *representation* 758b⟩≡
```
ASSERT TAG0.CastToAtom.Code = InteLex.ListType;
LType.CREATE;
ASSERT TAG1.CastToAtom.Code = InteLex.QualId;
LType.SetTrueClass (DuraCfg.GetClass (TAG1;
FOR i ∈ 2 .. TAG.Size - 1 DO
  ASSERT TAGi.CastToAtom.Code = InteLex.QualId;
  LType.AppendActual (DuraCfg.GetClass (TAGi;
  END;
RESULT ← LType;
```

When recognizing an attribute, two cases must be taken into account:

- If the `Component` we recognize is an `Atom` it can only refer to `InterThis` (See page 740).
- If it is a `Compound`, it refers to a conventional attribute that can perfectly parse itself (See method *DuraIntermediateFormat.InterAttribute.Parse* on page 738).

759a ⟨*Recognizing an attribute described by an* `Atom` *or a* `Compound` 759a⟩≡

```
METHOD ParseAttribute (Comp: Component): InterAttribute;
  BEGIN
  WHAT Comp OF
    IN Atom:
      ASSERT TAG.Code = InteLex.This;
      RESULT ← InterThis;
      END;
    IN Compound:
      RESULT.CREATE;
      RESULT.Parse (TAG);
      END;
    END;
  END ParseAttribute;
```

Recognizing an expression is slightly more complex: depending on the first atom of the `Compound`, a different `InterExpression`-derived instance must be created by calling the ad hoc creation method, and then parsed.

759b ⟨*Recognizing an expression described by a* `Compound` 759b⟩≡

```
⟨Various creation methods 761⟩

METHOD ParseExpression (Comp: Compound): InterExpression;
  VAR
    FirstCode: INTEGER;
  BEGIN
  FirstCode ← Comp_0.AtomCode;
  IF FirstCode = InteLex.Assign THEN
    ⟨A target for assignment must be taken into account 760a⟩
   ELSE
    CASE FirstCode OF
      ⟨Depending on the first token, create the proper InterExpression 760b⟩
      END;
    ⟨Polymorphic call to Parse 762a⟩
    END;
  END ParseExpression;
```

If the leading atom denotes an assignment, we perform a recursive call to get the actual expression, and then parse the target attribute for the expression.

760a ⟨*A target for assignment must be taken into account* 760a⟩≡

```
ASSERT Comp.Size = 3;
RESULT ← ParseExpression(Comp_2;
RESULT.SetTarget (ParseAttribute (Comp_1;
```

760b ⟨*Depending on the first token, create the proper* InterExpression 760b⟩≡

```
InteLex.Terminal:
  RESULT ← CreateTerminal;
  END;
InteLex.TerminalSet:
  RESULT ← CreateTerminalSet;
  END;
InteLex.NonTerminal:
  RESULT ← CreateNonTerminal;
  END;
InteLex.Concat, InteLex.Or:
  RESULT ← CreateAggregate;
  END;
InteLex.Repeat0, InteLex.Repeat1:
  RESULT ← CreateRepetition;
  END;
InteLex.Optional:
  RESULT ← CreateOptional;
  END;
InteLex.Case:
  RESULT ← CreateCase;
  END;
```

Such a list of trivial methods comes from a shortcoming in YAFL operator policy. Instead of making the creation of a new instance of a class an operator such as `new` in C++, it has been defined as a pseudo-method. Hence, one cannot create a new instance of a class in YAFL without defining a variable (local, attribute or result of a method) of this class. The methods listed here have no other purpose than to create new instances of specific classes.

761 ⟨*Various creation methods* 761⟩≡

```
METHOD CreateTerminal: InterTerminalExpression;
  BEGIN
  RESULT.CREATE;
  END CreateTerminal;

METHOD CreateTerminalSet: InterTerminalSetExpression;
  BEGIN
  RESULT.CREATE;
  END CreateTerminalSet;

METHOD CreateNonTerminal: InterNonTerminalExpression;
  BEGIN
  RESULT.CREATE;
  END CreateNonTerminal;

METHOD CreateAggregate: InterAggregateExpression;
  BEGIN
  RESULT.CREATE;
  END CreateAggregate;

METHOD CreateRepetition: InterRepetitionExpression;
  BEGIN
  RESULT.CREATE;
  END CreateRepetition;

METHOD CreateOptional: InterOptionalExpression;
  BEGIN
  RESULT.CREATE;
  END CreateOptional;

METHOD CreateCase: InterCaseExpression;
  BEGIN
  RESULT.CREATE;
  END CreateCase;
```

We now call the generic `Parse` method as defined in `InterExpression` (See page 704) and redefined in each of `InterExpression`-derived classes.

762a ⟨*Polymorphic call to* `Parse` 762a⟩≡

```
RESULT.Parse(Comp);
```

7.20.8 Import lists

762b ⟨*Definition import list* 762b⟩≡

```
IMPORT ArrayList;
FROM DuraFlat IMPORT FlatRuleComponent, FlatNonTerminal,
                     FlatTerminalSet,
                     FlatRHS, TerminalSet;
FROM DuraClDefinition IMPORT DuraClassDefinition;
FROM DuraGrClause IMPORT GrammarClause, GrammarRule;
FROM DuraLisp IMPORT Component, Compound, Atom;
FROM DuraOutput IMPORT DuraOutputStream;
FROM DuraSpecification IMPORT SyntacticSpecification;
IMPORT Space;
FROM Streams IMPORT InputStream, OutputStream;
FROM YaflNTList IMPORT NTList;
```

762c ⟨*Implementation import list* 762c⟩≡

```
FROM DuraCfg IMPORT DuraCfg;
FROM DuraClDefinition IMPORT DuraClassDefinition;
FROM DuraFlat IMPORT FlatRule, FlatTerminal, FlatNonTerminal;
FROM DuraLisp IMPORT Compound, Atom, Component;
FROM DuraModules IMPORT DuraDefinitionModule;
FROM DuraReductions IMPORT FetchNonTerminalReduction,
                           ConcatListReduction,
                           IndicatorReduction;
FROM YaflCfg IMPORT CurrentSpot;
FROM YaflModules IMPORT DefinitionModule;
FROM YaflPredefined IMPORT PredefItems;

FROM Conversions IMPORT IntConversions;
IMPORT ModuleTable;
IMPORT InteLex;
FROM Streams IMPORT StdOut, StdErr;
IMPORT String;
IMPORT SYSTEM;
```

7.21 DuraIntDictionary

The purpose of the `DuraIntDictionary` module is to provide facilities to store and retrieve arrays of integers of various array levels (that is, arrays, arrays of arrays, etc ...) in an simple and convenient way.

⟨*Definition* ⟩≡

```
DEFINITION MODULE DuraIntDictionary;

   ⟨IntDictionary class definition 768a⟩

END DuraIntDictionary;
```

⟨*Implementation* ⟩≡

```
IMPLEMENTATION MODULE DuraIntDictionary;

FROM Conversions IMPORT IntConversions;
FROM List IMPORT StringElement, StringList;
FROM Streams IMPORT InputStream, OutputStream, StdErr, StdOut;
IMPORT String;
IMPORT SYSTEM;

   ⟨IntDictionary class implementation 768b⟩
   ⟨Name class implementation 786⟩
   ⟨Tree class implementation 793⟩
   ⟨TreeElement class implementation 787⟩

END DuraIntDictionary;
```

7.21.1 IntDictionary

An `IntDictionary` can be defined formally as:

IntDictionary

$ErrorValue : \mathbb{Z}$

$TheArrs : iseq \operatorname{seq} \mathbb{Z}$

As opposed to "$\mapsto$" that denotes a total function, "$\pfun$" denotes a partial function, that is, a function which domain is a subset of the domain specified on the left of the operator. In this context, this means that the $Names$ function will yield a significant return value for a subset of all $STRING$s.

$TheNames : STRING$ (See page 977) $\pfun \mathbb{Z}$

$TheArrs$ denotes a unique ($iseq$) sequence of sequences of integers, while $Names$ implements the naming facility, that can associate a name with an entry in the dictionnary.

INIT

$TheArrs = \langle\,\rangle$
$TheNames = \{\,\}$

$StoreArray$ (See page 782) returns the negated value of the index where $Arr_1?$ has been stored within the dictionnary. The uniqueness property as defined above ensures that the same array cannot be stored more than once, or, more precisely, that storing two times the same array will result in a single storage, and $RESULT!$ being the same in both cases. The negated value is used so that in our applications where most of the integers we are dealing with are positive, return values from $StoreArray$ can be recognized conveniently as such.

$StoreArray$

The following clause states that this operation will not change any of $self$'s attributes, except $TheArrs$.

$\Delta(TheArrs)$
$Arr_1? : \text{seq}\,\mathbb{Z}$
$RESULT! : \mathbb{Z}$

$Arr_1? = \langle\,\rangle \Leftrightarrow RESULT! = 0$

The following clause states that if there was no sequence with value $Arr_1?$ in $TheArrs$ *before* invoking this operation, an entry with the value $Arr_1?$ is added to $TheArrs$ using list concatenation ($\frown$), and stored in $TheArrs$ made available *after* the operation is completed. Otherwise, the negated value of the index where value Arr_1 is found was stored is returned.

$$Arr_1? \notin \text{ran}\,TheArrs \Leftrightarrow TheArrs' = TheArrs \frown \langle Arr_1? \rangle \land RESULT! = -\#\,TheArrs'$$
$$Arr_1? \in \text{ran}\,TheArrs \Rightarrow TheArrs' = TheArrs \land TheArrs(-RESULT!) = Arr_1?$$

$StoreArrayArray$ (See page 783) is similar to $StoreArray$, except for the fact that it takes a sequence of sequence of integers as input. It first stores all the individual arrays using $StoreArray$ as described above, retains the negated indexes returned by $StoreArray$, and finally stores the array made of these negated indexes as any other array, again using $StoreArray$.

$StoreArrayArray$

$\Delta(TheArrs)$
$Arr_2? : \mathrm{seq}\,\mathrm{seq}\,\mathbb{Z}$
$RESULT! : \mathbb{Z}$

$Arr_2? = \langle\,\rangle \Leftrightarrow RESULT! = 0$
$s \in \mathrm{seq}\,\mathbb{Z}$
$\#s = \#Arr_2?$

The following clause is a multiple expansion of operation $StoreArray$, indexed by i.
"$StoreArray_i \mid Arr_1?_i = Arr_2?(i), RESULT!_i = s(i)$" indicates that on the i^{th} expansion, the value $Arr_2(i)$ should be substitued for parameter $Arr_1?$, and $s(i)$ should be substituted for $RESULT!$. In other words, $s(i)$ receives the value returned by operation $StoreArray$ when given $Arr_2?(i)$ as parameter.

$\forall\, i \in \mathrm{dom}\, Arr_2? \bullet StoreArray_i \mid Arr_1?_i = Arr_2?(i), RESULT!_i = s(i)$

Finally, we expand $StoreArray$ once more with s as parameter, and we don't specify any renaming clause for $RESULT!$ since the return value of this invocation of $StoreArray$ is to be used as the return value for $StoreArrayArray$.

$StoreArray \mid Arr_1? = s$

$GetArray$ (See page 784) simply returns an entry that was stored in the dictionary

$GetArray$

$Index? : \mathbb{Z}$
$RESULT! : \mathrm{seq}\,\mathbb{Z}$

$-Index? \in \mathrm{dom}\, TheArrs \cup \{0\}$
$Index? = 0 \Leftrightarrow RESULT! = \langle\,\rangle$
$Index? < 0 \Leftrightarrow RESULT! = TheArrs(-Index?)$

$GetArrayArray$ (See page 784) returns a sequence of sequences, by first taking a simple array, and by building the two-dimensional array made of the arrays one can get by applying $GetArray$ onto individual elements of this first array.

$GetArrayArray$

$Index? : \mathbb{Z}$
$RESULT! : \text{seq seq } \mathbb{Z}$

$-Index? \in \text{dom } TheArrs \cup \{0\}$
$Index? = 0 \Leftrightarrow RESULT! = \langle \rangle$
$Index? < 0 \Leftrightarrow \forall\, i \in \text{dom } TheArrs(-Index?) \bullet$
$\quad RESULT!(i) = TheArrs(-TheArrs(-Index?)(i))$

In theory, we could continue endlessly to generalize this to higher degrees ($StoreArrayArrayArray$, $GetArrayArrayArray$, etc.) but it has not proven useful for any practical purpose so far.

$SaveName$ (See page 785) and $FindName$ (See page 785) implement the naming scheme, where one can name a given entry in the $IntDictionary$.

$SaveName$

$\Delta(TheNames)$
$AName? : STRING$
$Value? : \mathbb{Z}$

$-Value? \in \text{dom } TheArrs$
$AName? \notin \text{dom } TheNames$

The precondition above states that before invoking $SaveName$, the $TheNames$ function did not yield any significant return value when provided $AName?$ as formal parameter.
The constraints hereunder states that a couple mapping $AName?$ onto $Value?$ is added to the partial function $TheNames$ as an effect of this operation.

$TheNames' = TheNames \cup \{AName? \mapsto Value?\}$

$FindName$

$AName? : STRING$
$RESULT! : \mathbb{Z}$

$AName? \notin \text{dom } Names \Leftrightarrow RESULT! = ErrorValue$
$AName? \in \text{dom } Names \Leftrightarrow RESULT! = Names(AName?)$

This reasonably formal specification has limitations, but shows clearly the basic idea: we keep a dictionary of *unique* arrays of integers, and give an index number to each of these arrays. When we wish to store an array of arrays, we merely store an array made of the indexes of each of the sub-arrays.

The whole idea is to make the $TheArrs$ attribute a *unique* sequence, which, formally at least, ensures that a given array cannot be referenced more than once. How we will guarantee that in practice is an entire different story.

This class also provides two additional facilities, namely the ability to name a given entry by means of the $SaveName$ operation, and to retrieve the named entry by means of the $FindName$ operation.

768a ⟨IntDictionary *class definition* 768a⟩≡

```
CLASS IntDictionary;

   CONST
      ErrorValue = -17891789;    -- Unlikely index in an array...

   REDEFINE METHOD CREATE;
   ⟨Saving and retrieving an IntDictionary, definition 770c⟩
   ⟨Storing arrays, definition 781b⟩
   ⟨Name management, definition 785a⟩

   ⟨Retrieving arrays, definition 783c⟩

   ⟨Treble management, definition 769d⟩

END IntDictionary;
```

768b ⟨IntDictionary *class implementation* 768b⟩≡

```
CLASS IntDictionary;

   ⟨Attributes 769a⟩
   ⟨Treble management, implementation 770a⟩
   ⟨Creating an IntDictionary 770b⟩

   ⟨Saving and retrieving an IntDictionary, implementation 771⟩
   ⟨Storing arrays, implementation 782a⟩
   ⟨Retrieving arrays, implementation 783d⟩
   ⟨Name management, implementation 785b⟩

END IntDictionary;
```

$\mathit{TheArrs}$ as described in the formal definition is a sequence of sequences of integers ($\mathit{TheAttrs} : \mathit{iseq}\,\mathrm{seq}\,\mathbb{Z}$). It is implemented here as an array of array of integers `TheArrs`, of which the first dimension is resized if required, and an integer attribute `TheUsedArrs` that indicates the number of entries that have been used so far.

769a ⟨*Attributes* 769a⟩≡

```
VAR
  TheArrs: ARRAY OF ARRAY OF INTEGER;
  TheUsedArrs: INTEGER;
```

`TheNameList` implements the naming facility (see the Names state attribute, and the $\mathit{SaveName}$ and $\mathit{FindName}$ operations in the formal definition) using the `Name` class (See page 785).

769b ⟨*Attributes* 769a⟩+≡

```
TheNameList: StringList(Name);
```

`TheTree` does not appear in the formal definition. It is the attribute that will implement the uniqueness (iseq) property in $\mathit{TheAttrs} : \mathit{iseq}\,\mathrm{seq}\,\mathbb{Z}$.

769c ⟨*Attributes* 769a⟩+≡

```
TheTree: Tree;
```

Even if the file where an `IntDictionary` is saved is pretty grumbled, for some applications, we would like to be sure that it cannot be decoded easily. Therefore, we provide an optional *treble factor* that will be used to XOR all integers read an written within an `IntDictionary` (See method *DuraIntDictionary.IntDictionary.Save* on the next page).

769d ⟨*Treble management, definition* 769d⟩≡

```
METHOD SetTreble (Treble: INTEGER);
  POST
    THIS.Treble = Treble;

METHOD Treble: INTEGER;
  POST
    NO_CHANGE;
```

Well, beloved reader, don't think you have a world-class secret here. As soon as this document is published, the value below will be changed.

770a ⟨*Treble management, implementation* 770a⟩≡

```
CONST
  ConstantTreble = 17*19*7*3*11;
VAR
  TheTreble: INTEGER;

METHOD SetTreble (Treble: INTEGER);
  BEGIN
  TheTreble ← Treble;
  END SetTreble;

METHOD Treble: INTEGER;
  BEGIN
  RESULT ← TheTreble;
  END Treble;
```

Creating an `IntDictionary` simply creates whatever is needed:

770b ⟨*Creating an* `IntDictionary` 770b⟩≡

```
REDEFINE METHOD CREATE;
  CONST
    InitialSize = 1024;
  BEGIN
  TheArrs.CREATE (InitialSize);
  TheUsedArrs ← 1;
  TheTree.CREATE;
  TheNameList.CREATE;
  END CREATE;
```

7.21.2 Saving and retrieving an `IntDictionary`

770c ⟨*Saving and retrieving an* `IntDictionary`, *definition* 770c⟩≡

```
METHOD Save (FName: ARRAY OF CHAR);
METHOD Load (FName: ARRAY OF CHAR): BOOLEAN;
```

This is rather boring work, but it has to be done consistently if one wishes to get reasonable performance and portability. The files that are read and written by the `Load` and `Save` methods below are plain ASCII files to make them easier to manage (YAFL does not support binary files as part of its standard library anyway) and to debug.

Their syntax can be defined as follows:

- If a non-zero *treble factor* has been set, the first line gives an integer value (which is *not* the treble factor itself) that is required to decypher every integer that is going to be read, preceded by a "`!`" character. In absence of such a line, the treble factor is 0.
- First, save name associations
- Then, put an empty line as a separator
- Then, an integer indicates the number of entries (understand: arrays)
- Then, for each of these entries, the array is saved, starting by an integer indicating its length, followed by its content. If the treble factor is non-zero, the arrays are saved in permuted order rather than their normal order of appeareance.
- All characters other than digits, the dot and the dash character are ignored.
- A comment starting with an opening parenthese "(" is ignored until the corresponding closing parenthese ")", even if it contains digits
- If a non-zero treble factor has been set, a RLE[8] scheme is used to compress the file. A leading dollar "`$`" sign indicates that the immediately following number is a counter, that represents the number of repetition of the subsequently following number in the file. If the treble factor is zero, the arrays are listed in extenso.

771 ⟨*Saving and retrieving an* `IntDictionary`, *implementation* 771⟩≡

```
METHOD Save (FName: ARRAY OF CHAR);
  VAR
    Counter, Index, Step: INTEGER;
    Output: OutputStream;
    p: ARRAY OF INTEGER;

    ⟨Local method for saving 776a⟩

  BEGIN
  Output.CREATE;
  Output.Create (FName, Output.WriteAccess);
  ⟨First, output TheTreble if required 772a⟩
  ⟨Then, save names 772c⟩
  EmitInt (TheUsedArrs);
  Purge;
  Index ← -1;
  ⟨Actually output the data 773⟩
  Purge;
  Output.Close;
  END Save;
```

[8] Running Length Encoding, which replaces subsequences of integers with the same value by a counter and a single occurence of the value.

The treble factor is produced, after XORing it with a constant factor so that one cannot just take the first value of the file to decode the rest without knowing the constant value in the first place. It is not much of a protection, but it simply would be too simple to crack without it.

772a ⟨*First, output* `TheTreble` *if required* 772a⟩≡

```
IF TheTreble ≠ 0 THEN
  Output.WriteString (‘‘!”);
  Output.WriteInt (SYSTEM.BitXor(ConstantTreble, TheTreble), 0);
  Output.WriteLn;
  END;
```

The line with a leading “!” must be read and decoded as well, according to the way it was written.

772b ⟨*Check for reading treble factor* 772b⟩≡

```
ELSIF Line0 = ‘‘!” THEN
 TheTreble ← IntConversions.StringToInt (Line.SLICE (1, Line.SIZE - 1));
 TheTreble ← SYSTEM.BitXor (TheTreble, ConstantTreble);
```

772c ⟨*Then, save names* 772c⟩≡

```
FOR n ∈ TheNameList DO
  n.Save (Output, TheTreble);
  Purge;
  END;
Purge;
```

This is the empty line, to separate names from the rest of the dictionary’s data.

772d ⟨*Then, save names* 772c⟩+≡

```
Output.WriteLn;
```

For various reasons, we want to be able to store information and make it difficult for someone to find it back, and ultimately, to modify it. For instance, licence keys for some of our software are to be stored in the `IntDictionary` and we want to be able to track the origin of any copy of our software we find.

Similarly, a key that inhibits some of the system’s functionalities can be stored as well, and we would like to make the job of the hacker as difficult as possible. After all, if he (or she) wants to cheat, at least let him (or her) sweat.

The problem lies in the fact that this code is freely available, and if the hacker searches before starting to code, he (or she) might as well find the solution here ! Besides, to ensure portability, most of our applications are delivered with the intermediate C code generated by the YAFL compiler. Hence, given a bit of courage, one can reverse engineer the intermediate C code to understand how the dictionary file is structured. The output of the YAFL compiler is notoriously unreadable, and it goes through a supplementary shrouding pass to obfuscate it even further before distribution, but in theory, it is still possible.

One must also understand that no crucial zillion-dollar secret is to be saved here, and within the scope of the industrial implications of this work, it is not worth to start implementing some kind of complex scheme based on private and public keys, prime numbers with over 800 digits, etc. Our sole purpose is to make it harder for anyone to decode the structure of the `IntDictionary` file.

The obfucation of the text file is controlled by the *treble factor*, which is used to alter all integer values. Of course, such an alteration makes it harder to read since individual values get scrambled, but there are cases where one might be tempted to use mere position information (that is, knowing that some information at the beginning or rather at the end of the file) to try to change it, for instance, to remove some identification that might have been put into the file.

Hence, the treble factor is also used to reorder the data before it is stored on the output text file, so that someone who wishes to change data he (or she) supposes must be stored at the end of the tables cannot just try to change values at the end of the text file.

This reordering is performed by:

$$Reorder : 0 .. TheUsedArrs - 1 \mapsto 0 .. TheUsedArrs - 1$$

$$Reorder(i) = (i * Step) \bmod TheUsedArrs$$

[9]If $Step = 1$ we have: $\forall\, i \in 0 .. TheUsedArrs - 1 \bullet Reorder(i) = i$. Otherwise, we'll choose a value for $Step$ that is relatively prime with $TheUsedArrs$, and we'll have $\operatorname{dom} Reorder = \operatorname{ran} Reorder = 0 .. TheUsedArrs - 1$.

The following chunk computes `Step`, based on these premises (See method *DuraIntDictionary.IntDictionary.RelativePrime* on page 780):

773 ⟨*Actually output the data* 773⟩≡

```
IF TheTreble = 0 THEN
  Step ← 1;
 ELSE
  Step ← RelativePrime (TheUsedArrs);
  END;
```

[9]$Reorder$ as presented here should be understood as a communication tool, it is not implemented as a YAFL method.

We then go through all the arrays stored in `THIS`, using `Step` to reorder them appropriately.

774a ⟨*Actually output the data* 773⟩+≡

```
FOR i ∈ 1 .. TheUsedArrs DO
  Index ← (Index + Step) MOD TheUsedArrs;
  IF Index = 0 THEN
    ⟨Emit entry 0 775a⟩
   ELSE
    p ← TheArrs_Index;
    IF (TheTreble = 0) THEN
      ⟨Emit a comment indicating the new entry in the file 775b⟩
      END;
    ASSERT Index ≠ 0;
    ⟨Emit array p 775c⟩
    END;
  END;
```

Quite predictably, reading the data mimics writing the data quite closely . . .

774b ⟨*Actually read the data* 774b⟩≡

```
IF TheTreble = 0 THEN
  Step ← 1;
 ELSE
  Step ← RelativePrime (TheUsedArrs);
  END;
FOR i ∈ 1 .. TheUsedArrs DO
  Index ← (Index + Step) MOD TheUsedArrs;
  ASSERT TheArrs_Index = Λ;
  ⟨Read the actual array 774c⟩
  END;
```

To read an array, we must first read its size, and allocate the array and read it if this size is nonzero, which is the case for entry 0 in our array of arrays.

774c ⟨*Read the actual array* 774c⟩≡

```
NewSize ← AcceptInt;
IF NewSize > 0 THEN
  TheArrs_Index.CREATE (NewSize);
  FOR j ∈ 0 .. NewSize - 1 DO
    TheArrs_Index,j ← AcceptInt;
    END;
  END;
```

Since entry 0 is not actually allocated in the dictionary, we must consider it as a special case, and write the corresponding entry on the output file manually, producing a fully documented entry if no treble factor has been specified.

775a ⟨*Emit entry 0* 775a⟩≡
```
IF TheTreble = 0 THEN
    Output.WriteLine (" ( Entry: 0 Size: 0 ) 0");
  ELSE
    EmitInt (0);
    END;
```

775b ⟨*Emit a comment indicating the new entry in the file* 775b⟩≡
```
Output.WriteString (" ( Entry: ");
Output.WriteInt (i, 0);
Output.WriteString (" Size: ");
Output.WriteInt (p.SIZE, 0);
Output.WriteLine (" )");
```

775c ⟨*Emit array* p 775c⟩≡
```
IF (TheTreble = 0) THEN
    Purge;
    Output.WriteLn;
    END;
EmitInt (p.SIZE);
FOR x ∈ p DO
    EmitInt (x);
    END;
IF (TheTreble = 0) THEN
    Purge;
    Output.WriteLn;
    END;
```

The various chunks above call `EmitInt` and `Purge` to produce integers on the output file. The following chunks encapsulate additional treatments that are performed before actually writing these integers. `LocalEmitInt` keeps a counter so that each line is truncated before it gets too long to be edited conveniently, and outputs the integer taking the treble factor into account if defined.

It is worth noticing that calling `SYSTEM.BitXor` does not result in a true function call with its performance penalty. [31] explains in detail how such low-level facilities are inlined, thereby providing performance equivalent to what can be achieved using C operators.

776a ⟨*Local method for saving* 776a⟩≡

```
METHOD LocalEmitInt(i: INTEGER);
  BEGIN
  Counter ← Counter + 1;
  IF Counter MOD 10 = 0 THEN
    Output.WriteLn;
    END;
  IF TheTreble = 0 THEN
    Output.WriteInt (i,0);
   ELSE
    Output.WriteInt (SYSTEM.BitXor (i, TheTreble),0);
    END;
  Output.WriteChar (“ ”);
  END LocalEmitInt;
```

`Purge` actually generates the integer, with a leading dollar “$” and a count to implement RLE if the counter is greater than 1.

776b ⟨*Local method for saving* 776a⟩+≡

```
VAR
  Count, Value: INTEGER;

METHOD Purge;
  BEGIN
  IF Count = 1 THEN
    LocalEmitInt (Value);
   ELSIF Count > 1 THEN
    Output.WriteChar (“ ”);
    Output.WriteChar (“$”);
    Output.WriteChar (“ ”);
    LocalEmitInt (Count);
    LocalEmitInt (Value);
    END;
  Count ← 0;
  Value ← 0;
  END Purge;
```

`EmitInt` take an integer to output, increment the counter if the integer is the same as the previous one, or calls `Purge` and starts a new sequence.

777a ⟨*Local method for saving* 776a⟩+≡

```
METHOD EmitInt(i: INTEGER);
  BEGIN
  IF (TheTreble ≠ 0) ∧ (Value = i) ∧ (Count > 0) THEN
    Count ← Count + 1;
   ELSE
    Purge;
    Count ← 1;
    Value ← i;
    END;
  END EmitInt;
```

As expected, the method that retrieves an `IntDictionary` mimics the methods that saves it quite closely.

777b ⟨*Saving and retrieving an* `IntDictionary`*, implementation* 771⟩+≡

```
METHOD Load (FName: ARRAY OF CHAR): BOOLEAN;
  VAR
    Line: ARRAY OF CHAR;
    GoOn: BOOLEAN;
    Input: InputStream;
    Step, Index, NewSize: INTEGER;
    Words: ARRAY OF ARRAY OF CHAR;

    ⟨Local methods for Load 778b⟩

  BEGIN
  Input.CREATE;
  Input.Open(FName, Input.ReadAccess);
  IF Input.ErrorCode = Input.NoError THEN
    ⟨Read names and treble 778a⟩
    TheUsedArrs ← AcceptInt;
    TheArrs.CREATE (TheUsedArrs);
    Index ← -1;
    ⟨Actually read the data 774b⟩
    Input.Close;
    RESULT ← TRUE;
    END;
  END Load;
```

Reading names means reading the input until the empty line that serves as separator is encountered. The `BreakInWords` method provided by YAFL's standard library module `String` is used to perform basic parsing on input lines. YAFL

778a ⟨*Read names and treble* 778a⟩≡

```
GoOn ← TRUE;
WHILE GoOn DO
  Line ← String.Strip(Input.ReadLine);
  IF (Line = Λ) ∨ (Line.SIZE = 0) THEN
    GoOn ← FALSE;
   ⟨Check for reading treble factor 772b⟩
   ELSE
    Words ← String.BreakInWords (Line, " ", Λ);
    ASSERT Words.SIZE = 2;
    SaveName(Words_1, SYSTEM.BitXor (TheTreble,
                                  IntConversions.StringToInt(Words_0)))
    END;
  END;
```

When accepting an integer from the input, we rather write brute force code instead of using the standard library, which would be needlessly more versatile and would require at least one additional allocation for every integer that is being read.

778b ⟨*Local methods for* `Load` 778b⟩≡

```
METHOD LocalAcceptInt(LookAhead: CHAR): INTEGER;
  VAR
    Ch: CHAR;
    Negative: BOOLEAN;
  BEGIN
  Ch ← LookAhead;
  ASSERT ((Ch ≥ "0") ∧ (Ch ≤ "9")) ∨ (Ch = "-");
  IF Ch = "-" THEN
    Negative ← TRUE;
    Ch ← Input.ReadChar;
    END;
  WHILE (Ch ≥ "0") ∧ (Ch ≤ "9") DO
    RESULT ← RESULT * 10 + SYSTEM.ORD(Ch) - SYSTEM.ORD("0");
    Ch ← Input.ReadChar;
    END;
  IF Negative THEN
    RESULT ← - RESULT;
    END;
  IF TheTreble ≠ 0 THEN
    RESULT ← SYSTEM.BitXor(TheTreble, RESULT);
    END;
  END LocalAcceptInt;
```

`AcceptInt` implements RLE for reading. When reading a counter, the value is decremented for the current invocation of `AcceptInt`. Otherwise, if a counter of 3 is specified, `AcceptInt` would return the same value 4 times rather than 3.

779 ⟨*Local methods for* `Load` 778b⟩+≡

```
VAR
  Count, Value: INTEGER;

METHOD AcceptInt: INTEGER;
  VAR
    Ch: CHAR;
  BEGIN
  IF Count > 0 THEN
    RESULT ← Value;
    Count ← Count - 1;
    IF Count = 0 THEN
      Value ← 0;
      END;
   ELSE
    Ch ← Skip;
    IF Ch = ‘‘$” THEN
      Ch ← Skip;
      Count ← LocalAcceptInt(Ch)-1;
      Ch ← Skip;
      Value ← LocalAcceptInt (Ch);
      RESULT ← Value;
     ELSE
      RESULT ← LocalAcceptInt (Ch);
      END;
    END;
  END AcceptInt;
```

and `Skip` just skips white space and comments.

780a ⟨*Local methods for* `Load` 778b⟩+≡
```
METHOD Skip: CHAR;
  VAR
    GoOn: BOOLEAN;
  BEGIN
  RESULT ← Input.ReadChar;
  GoOn ← TRUE;
  WHILE GoOn ∧ ¬ Input.Eof DO
    CASE RESULT OF
      ‘‘0” .. ‘‘9”, ‘‘-”, ‘‘$”:
        GoOn ← FALSE;
        END;
      ‘‘(”:
        ⟨Skip comments 780b⟩
        END;
     ELSE
      RESULT ← Input.ReadChar;
      END;
    END;
  END Skip;
```

By construction, comments never contains parentheses. Hence, when skipping comments, we don't have to consider possible parenthese nesting.

780b ⟨*Skip comments* 780b⟩≡
```
WHILE (RESULT ≠ ‘‘)”) ∧ ¬ Input.Eof DO
  RESULT ← Input.ReadChar;
  END;
```

`RelativePrime` returns an integer which is relatively prime with the `v` parameter it receives. It considers three predefined values, namely, 97, 101, 103, and takes the first one which is relatively prime with `v`. This method is based on the fact that no `v` smaller than $97 * 101 * 103$ can be a simultaneous multiple of 97, 101 and 103, and this is an acceptable upper bound as far as the predictable applications of `IntDictionary` are concerned.

Of course, one might argue that the number of values to test is limited, but here again, as soon as this document is published, the values listed here will be changed. In any case, we are fully aware of the limited security our scheme actually guarantees, but it is good enough for our current purposes.

781a ⟨*Saving and retrieving an* IntDictionary, *implementation* 771⟩+≡
```
METHOD RelativePrime (v: INTEGER): INTEGER;
  VAR
    t: ONCE ARRAY OF INTEGER;
  BEGIN
  IF t = Λ THEN
    t.CREATE (3);
    t₀ ← 97;
    t₁ ← 101;
    t₂ ← 103;
    END;
  RESULT ← FIRST x ∈ t • v MOD x ≠ 0;
  END RelativePrime;
```

7.21.3 Storing arrays and ensuring uniqueness

The post-conditions of these three methods are implied by their specification, that indicates that they return a negated index into an array.

781b ⟨*Storing arrays, definition* 781b⟩≡
```
METHOD StoreArray (Arr: ARRAY OF INTEGER): INTEGER;
  POST
    RESULT ≤ 0;

METHOD StoreArrayArray (Arr: ARRAY OF
                             ARRAY OF INTEGER): INTEGER;
  POST
    RESULT ≤ 0;

METHOD StoreArrayArrayArray (Arr: ARRAY OF
                                  ARRAY OF
                                  ARRAY OF INTEGER): INTEGER;
  POST
    RESULT ≤ 0;
```

We use `TheTree` (See page 769) and the `Find` method (See page 793) to first check whether the array `Arr` has already been stored or not. If it has been stored, we return the already stored array's negated index. Otherwise, we store `Arr` and return the newly allocated negated index.

782a ⟨*Storing arrays, implementation* 782a⟩≡

```
METHOD StoreArray (Arr: ARRAY OF INTEGER): INTEGER;
VAR
  Temp: ARRAY OF ARRAY OF INTEGER;
  t: TreeElement;
BEGIN
IF (Arr ≠ Λ) ∧ (Arr.SIZE > 0) THEN
  t ← TheTree.Find (Arr);
  IF t ≠ Λ THEN
    RESULT ← −t.IndexNr;
   ELSE
    ⟨Arr has not been stored yet, add it 782b⟩
    END;
  END;
ASSERT RESULT ≤ 0;
END StoreArray;
```

Before actually adding `Arr`, we first have to ensure that `TheArrs` is big enough, and resize to the double of its current size if it is full.

782b ⟨`Arr` *has not been stored yet, add it* 782b⟩≡

```
IF TheUsedArrs ≥ TheArrs.SIZE THEN
  Temp.CREATE (TheUsedArrs);
  TheArrs ← TheArrs + Temp;
  END;
```

We then add `Arr` in `TheArrs`...

782c ⟨`Arr` *has not been stored yet, add it* 782b⟩+≡

```
TheArrs_TheUsedArrs ← Arr;
RESULT ← −TheUsedArrs;
TheUsedArrs ← TheUsedArrs + 1;
```

Then, we CREATE a new `TreeNode` instance, and store it in `TheTree` to ensure that further lookups for the same array will find it.

782d ⟨`Arr` *has not been stored yet, add it* 782b⟩+≡

```
t.CREATE (Arr, TheUsedArrs−1);
TheTree.Store (t);
```

Based on the `StoreArray` method described above, `StoreArrayArray` and `StoreArrayArrayArray` are pretty straightforward operations. We use the ability to represent an array of integer by a single integer (namely, the corresponding entry number in the dictionary) to store arrays of various nesting levels in the same dictionary.

783a ⟨*Storing arrays, implementation* 782a⟩+≡

```
METHOD StoreArrayArray (Arr: ARRAY OF ARRAY OF INTEGER): INTEGER;
  VAR
    p: ARRAY OF INTEGER;
  BEGIN
  p.CREATE (Arr.SIZE);
  FOR i ∈ 0 .. p.SIZE - 1 DO
    p_i ← StoreArray(Arr_i);
    END;
  RESULT ← StoreArray(p);
  END StoreArrayArray;
```

783b ⟨*Storing arrays, implementation* 782a⟩+≡

```
METHOD StoreArrayArrayArray
              (Arr: ARRAY OF
                    ARRAY OF
                    ARRAY OF INTEGER): INTEGER;
  VAR
    p: ARRAY OF INTEGER;
  BEGIN
  p.CREATE (Arr.SIZE);
  FOR i ∈ 0 .. p.SIZE - 1 DO
    p_i ← StoreArrayArray(Arr_i);
    END;
  RESULT ← StoreArray(p);
  END StoreArrayArrayArray;
```

783c ⟨*Retrieving arrays, definition* 783c⟩≡

```
METHOD GetArray (Index: INTEGER): ARRAY OF INTEGER;
METHOD GetArrayArray (Index: INTEGER): ARRAY OF
                                       ARRAY OF INTEGER;

METHOD MainArray: ARRAY OF ARRAY OF INTEGER;
```

For various optimization purposes, we give direct access to the class internal structure. It is then our responsibility not to jeopardize its integrity . . .

783d ⟨*Retrieving arrays, implementation* 783d⟩≡

```
METHOD MainArray: ARRAY OF ARRAY OF INTEGER;
  BEGIN
  RESULT ← TheArrs;
  END MainArray;
```

Since `StoreArray` and its cousins return negative value, we must negate them again to extract the result value from `TheArrs`.

784a ⟨*Retrieving arrays, implementation* 783d⟩+≡
```
METHOD GetArray (Index: INTEGER): ARRAY OF INTEGER;
  BEGIN
  ASSERT -Index ≥ 0;
  ASSERT -Index < TheArrs.SIZE;
  RESULT ← TheArrs_{-Index};
  END GetArray;
```

When returning two-dimensional arrays, we keep a copy of the returned array referenced by `TheArrsArrs`, so that further queries with the same index will not require yet another allocation and computation.

784b ⟨*Retrieving arrays, implementation* 783d⟩+≡
```
VAR
  TheArrsArrs: ARRAY OF ARRAY OF ARRAY OF INTEGER;

METHOD GetArrayArray (Index: INTEGER): ARRAY OF
                                       ARRAY OF INTEGER;
  VAR
    Model: ARRAY OF INTEGER;
    p: ARRAY OF ARRAY OF INTEGER;
  BEGIN
  ASSERT Index ≤ 0;
  IF Index ≠ 0 THEN
    IF TheArrsArrs = Λ THEN
      TheArrsArrs.CREATE (TheArrs.SIZE);
      END;
    IF TheArrsArrs_{-Index} = Λ THEN
      Model ← GetArray(Index);
      IF Model ≠ Λ THEN
        p.CREATE (Model.SIZE);
        FOR j ∈ 0 .. p.SIZE - 1 DO
          p_j ← GetArray(Model_j);
          END;
        END;
      TheArrsArrs_{-Index} ← p;
      END;
    RESULT ← TheArrsArrs_{-Index};
    END;
  END GetArrayArray;
```

Name management as provided by `IntDictionary` is merely a matter of attaching associations between character strings and integers, that can be saved and retrieved when saving and retrieving an `IntDictionary`. It is clear that the original idea was to be able to name arrays stored by the `StoreArray` method, but it is not enforced in any way.

Name management is based on the `Name` class (See the current page). Since there should not be more than a dozen of such names attached to a given `IntDictionary`, linear search is considered good enough.

785a ⟨*Name management, definition* 785a⟩≡

```
METHOD SaveName (AName: ARRAY OF CHAR;
            Value: INTEGER);
METHOD FindName (AName: ARRAY OF CHAR): INTEGER;
```

785b ⟨*Name management, implementation* 785b⟩≡

```
METHOD SaveName (AName: ARRAY OF CHAR;
                  Value: INTEGER);
  VAR
    TheName: Name;
  BEGIN
  ASSERT FindName (AName) = ErrorValue;
  TheName.CREATE (AName, Value);
  TheNameList.Append (TheName);
  END SaveName;
```

785c ⟨*Name management, implementation* 785b⟩+≡

```
METHOD FindName (AName: ARRAY OF CHAR): INTEGER;
  VAR
    p: Name;
  BEGIN
  p ← TheNameList.FindString (AName);
  IF p ≠ Λ THEN
    RESULT ← p.Val;
   ELSE
    RESULT ← ErrorValue;
    END;
  END FindName;
```

7.21.4 `Name` class

The `Name` class implements the association between a name, made of a character string, and an entry number in the dictionary. Highly uninteresting stuff...

This class might be a bit awkward to read, unless one knows that `StringElement` is all about. It is a rather simple class, that encapsulates a character string that can be given at creation time, assigned by using the `Set` method and consulted using the `Get` method. Lists that contain `StringElement` instances can be searched conveniently using that character string as search key.

786 ⟨`Name` *class implementation* 786⟩≡

```
CLASS Name;
  INHERITS StringElement;
  VAR
    TheVal: INTEGER;

  REDEFINE METHOD CREATE (Data: ARRAY OF CHAR;
                          Val: INTEGER);
    BEGIN
    BASE(Data);
    TheVal ← Val;
    END CREATE;

  METHOD Val: INTEGER;
    BEGIN
    RESULT ← TheVal;
    END Val;

  METHOD Save (Output: OutputStream;
               Treble: INTEGER);
    BEGIN
    IF Treble ≠ 0 THEN
      Output.WriteInt(SYSTEM.BitXor(TheVal, Treble), 0);
     ELSE
      Output.WriteInt(TheVal, 0);
      END;
    Output.WriteChar (‘‘ ”);
    Output.WriteLine (Get);
    END Save;

END Name;
```

7.21.5 `TreeElement` class

`TreeElement` implements an element that will be stored within a tree to be able to determine whether a given array of integers has already been stored in the dictionary or not.

The interest of this class lies in its impact on performance. From a strictly theoretical point of view, its existence is not even justified by a mention in the formal specification above. However, a naive implementation of the uniqueness property was far too slow to be used in a production environment.

Just as most tree structures, ours is recursive, and holds a left and right subtree. It also holds an index in `TheIndexNr` that indicates where the array of integer was stored in the dictionary.

787 ⟨`TreeElement` *class implementation* 787⟩≡

```
CLASS TreeElement;

   VAR
     Data: ARRAY OF INTEGER;
     TheIndexNr: INTEGER;
     Left, Right: TreeElement;

   CONST
     IsSmaller = -1;
     IsEqual = 0;
     IsGreater = 1;

   REDEFINE METHOD CREATE (Data: ARRAY OF INTEGER;
                           TheIndexNr: INTEGER);
     BEGIN
     BASE;
     THIS.Data ← Data;
     THIS.TheIndexNr ← TheIndexNr;
     END CREATE;

   METHOD IndexNr: INTEGER;
     BEGIN
     RESULT ← TheIndexNr;
     END IndexNr;

   ⟨Storing a TreeElement in my subtree 788a⟩
   ⟨Finding a TreeElement starting at this 790c⟩
   ⟨Comparing two arrays of integers 791⟩
   ⟨Check to ensure that everything is consistent 792⟩

END TreeElement;
```

Storing a `TreeElement` is pretty straightforward: first check whether it must be stored on the left or on the right subtree. If the corresponding subtree is empty (understand: VOID), `t` is attached as initial subtree.

788a ⟨*Storing a* `TreeElement` *in my subtree* 788a⟩≡

```
METHOD Store(t: TreeElement);
  VAR
    Temp: TreeElement;
    Cmp: INTEGER;
  BEGIN
  Cmp ← CompArray (Data, t.Data);
  ⟨Ensure that the tree is consistent 788b⟩
  IF Cmp = IsGreater THEN
    IF Left = Λ THEN
      Left ← t;
     ELSE
      Left.Store (t);
      ⟨Ensuring balancing on the left side 790a⟩
      END;
   ELSIF Right = Λ THEN
    Right ← t;
   ELSE
    Right.Store (t);
    ⟨Ensuring balancing on the right side 790b⟩
    END;
  END Store;
```

By construction, we should never try to insert an array that has already been stored before in the tree.

788b ⟨*Ensure that the tree is consistent* 788b⟩≡

```
ASSERT Cmp ≠ IsEqual;
```

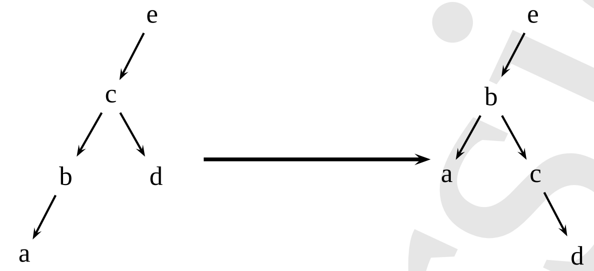
7.21.6 Keeping our trees balanced

The tree as managed here has only a very limited lifetime. On the other hand, experience with early prototypes has shown that with significant parsers, a large proportion of the entire run time was spent in searching this tree. A bit of analysis showed that long *sorted* sequences (or subsequences) were inserted, jeopardizing the benefits of the tree structure by making it dramatically unbalanced.

So, I first thought of the design shown in figure 7.12[10], where node b has no right subtree. The tree was then balanced by promoting the b node, reducing the path to a. This scheme gave rather good results, and I was quite pleased then.

[10] All the examples are shown as they apply to the balancing of the left subtree; one can easily extrapolate them to the right subtree.

Figure 7.12 Original balancing pattern

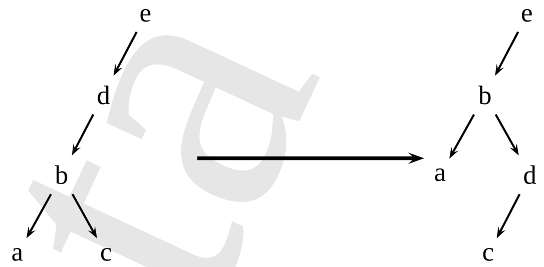


I was so pleased, that I fell into a tempting but dreadful trap: I explained this to several people, among who Jean-Christophe Real. As he usually does, he listened silently to my enthousiast explanations. I should have known better: he called me a few days afterward, to explain how my scheme was badly sub-optimal, and how it could be improved.

He noticed that my scheme suffered from a major drawback: by using the state of b as my sole criteria, I happen to *increase* instead of reduce the path length to c and d, even if d happens to be on top of a huge subtree. In such cases, by so-called balancing the tree, I made things even worse.

Jean-Christophe is not that kind of guy who find flaws in other people's systems without at least proposing an alternate scheme. His proposal is depicted in figure 7.13.

Figure 7.13 Vastly improved balancing pattern (Thanks, Jean-Christophe)



Instead of basing his criteria on what happens on b, he ensures that d has no right subtree. Hence, promoting b will increase the path length to d, but at least, we know for sure that this will have no further effect. By increasing this path length by one, and by reducing the path lengths to b and a, and to the entire subtree under a, he knows for sure that the worst case of his scheme remains reasonable.

Of course, the right thing to do would have been to check the literature about tree balancing algorithms in the first place, but it is one of those topics (sorting, binary search) which we all (well, me, at least) believe we know by heart and which proves to hold more mysteries than originally thought.

790a ⟨*Ensuring balancing on the left side* 790a⟩≡

```
IF (Left.Left ≠ Λ) ∧ (Left.Right = Λ) THEN
  Temp ← Left;                      -- Temp := d
  Left ← Left.Left;                 -- Left := b
  Temp.Left ← Left.Right;           -- d.Left := c
  Left.Right ← Temp;                -- b.Right := d;
  END;
```

Well, balancing on the right side is pathetically similar to balancing on the left side... I have to confess I basically retyped this without even trying to understand it again.

790b ⟨*Ensuring balancing on the right side* 790b⟩≡

```
IF (Right.Right ≠ Λ) ∧ (Right.Left = Λ) THEN
  Temp ← Right;
  Right ← Right.Right;
  Temp.Right ← Right.Left;
  Right.Left ← Temp;
  END;
```

Based on the way `TreeElements` are stored (see above), finding a `TreeElement`, given an array of integer `Arr` is pretty simple: if `Arr<Data`, search on the left subtree if present; if `Arr>Data`, search on the right subtree if present; otherwise (`Arr=Data`), well, `THIS` is the node we are looking for. `CompArray` (See the facing page) is used to compare arrays of integers.

790c ⟨*Finding a* `TreeElement` *starting at this* 790c⟩≡

```
METHOD Find(Arr: ARRAY OF INTEGER): TreeElement;
  BEGIN
  CASE CompArray (Arr, Data) OF
    IsSmaller:
      IF Left ≠ Λ THEN
        RESULT ← Left.Find (Arr);
        END;
      END;
    IsEqual:
      RESULT ← THIS;
      END;
    IsGreater:
      IF Right ≠ Λ THEN
        RESULT ← Right.Find (Arr);
        END;
      END;
    END;
  END Find;
```

We have no interest in lexical sorting of arrays, which would imply something as:

$$\{1,3,7\} < \{1,4,9\} < \{2\} < \{2,4\}$$

We just need an order relationship among arrays. We use the length as the most discriminant criterion, since it provides efficient comparison of arrays that would require long loops if lexical sorting had been used. We have:

$$\{2\} < \{2,4\} < \{1,3,7\} < \{1,4,9\}$$

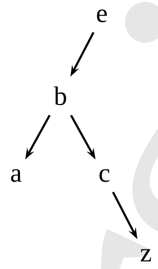
791 ⟨*Comparing two arrays of integers* 791⟩≡

```
METHOD CompArray (One, Two: ARRAY OF INTEGER): INTEGER;
  BEGIN
  IF One.SIZE > Two.SIZE THEN
    RESULT ← IsGreater;
   ELSIF One.SIZE < Two.SIZE THEN
    RESULT ← IsSmaller;
   ELSE
    RESULT ← IsEqual;
    FOR i ∈ 0 .. One.SIZE - 1 WHILE RESULT = IsEqual DO
     IF One_i > Two_i THEN
        RESULT ← IsGreater;
       ELSIF One_i < Two_i THEN
        RESULT ← IsSmaller;
        END;
      END;
    END;
  END CompArray;
```

Ensuring that a `TreeNode` is consistent means simply ensuring that its subtrees are consistent if present, and that the key on the right subtree is greater than `THIS`'s, and that the key on the left subtree is smaller than `THIS`'s.

This check is not perfect, and we might consider improving it, if we have troubles with the dictionaries. For instance, figure 7.14 depicts a tree which pass this check, while it clearly is not correct. In practice, even as imperfect as it is, this check detected major design flaws and we are pretty confident that the entire construction is rather solid now.

Figure 7.14 Undetected incorrect tree



792 ⟨Check *to ensure that everything is consistent* 792⟩≡

```
METHOD Check;
  BEGIN
  DEBUG
    IF Right ≠ Λ THEN
      Right.Check;
      ASSERT CompArray(Data, Right.Data) = IsSmaller;
      END;
    IF Left ≠ Λ THEN
      Left.Check;
      ASSERT CompArray(Data, Left.Data) = IsGreater;
      END;
    END;
  END Check;
```

7.21.7 `Tree` class

Well, based on a working `TreeElement` class, implementing a `Tree` is of course no big deal...

793 ⟨`Tree` *class implementation* 793⟩≡

```
CLASS Tree;

  VAR
    Root: TreeElement;

  METHOD Find(Arr: ARRAY OF INTEGER): TreeElement;
    BEGIN
    IF Root ≠ Λ THEN
      RESULT ← Root.Find(Arr);
      END;
    END Find;

  METHOD Store(t: TreeElement);
    BEGIN
    IF Root = Λ THEN
      Root ← t;
     ELSE
      Root.Store(t);
      DEBUG
        Root.Check;
        END;
      END;
    END Store;

END Tree;
```

7.22 DuraLex

YAFL

This module holds a `ONCE` class that takes care of the addition of new keywords to the original YAFL compiler. These keywords are used within grammar rules in YAFL classes that have been marked as being `NONTERMINAL` classes. The values of the keywords listed here is of no intrinsic importance, as long as they are all different from each other, and different from other keywords added to the YAFL compiler.

⟨*Definition* ⟩≡

```
DEFINITION MODULE DuraLex;

  ONCE CLASS DuraLex;

    CONST
      NonTerminal = 10081;
      Grammar = 10082;
      Precedence = 10083;
      LeftAssoc = 10084;
      RightAssoc = 10085;
      Snip = 10086;
      Recover = 10087;
      NoCut = 10088;
      NoRecover = 10089;

    METHOD EnterNewKeywords;
  END DuraLex;

END DuraLex;
```

YAFL

Adding a new token to the YAFL compiler happens by creating new instances of the `UserDefinedToken` class, giving the image of the token, and the code to return when the token has been found.

These new tokens will be usable when parsing extensions to YAFL (See class *DuraLookAhead.DuraStreamLookAhead* on page 809).

⟨*Implementation* ⟩≡

```
IMPLEMENTATION MODULE DuraLex;

FROM YaflLex IMPORT UserDefinedToken;

  ONCE CLASS DuraLex;

    METHOD EnterNewKeywords;
      VAR
        NewLex: UserDefinedToken;
      BEGIN
      NewLex.CREATE (“NONTERMINAL”, NonTerminal);
      NewLex.CREATE (“GRAMMAR”, Grammar);
      NewLex.CREATE (“PRECEDENCE”, Precedence);
      NewLex.CREATE (“LEFTASSOC”, LeftAssoc);
      NewLex.CREATE (“RIGHTASSOC”, RightAssoc);
      NewLex.CREATE (“SNIP”, Snip);
      NewLex.CREATE (“RECOVER”, Recover);
      NewLex.CREATE (“NOSNIP”, NoCut);
      NewLex.CREATE (“NORECOVER”, NoRecover);
      END EnterNewKeywords;

  END DuraLex;

END DuraLex;
```

7.23 DuraLisp

`DuraLisp` implements a primitive format that can be easily stored into and retrieved from disk files. It implements persistent storage for grammatical information. It is called `DuraLisp` because it is based on a simple dichotomy between *atoms* and *compounds*, which in turn contain atoms or other compounds. This construction allows convenient generic save and retrieve primitives. Given the entire scope of this work, I did not feel like writing yet another parser to parse yet another intermediate format. Of course, now that I have `DURA` available, writing such as parser is not such a big deal, I could use `DURA` to bootstrap some of its components. There is no urgent need which would justify this additional work now, since the intermediate format is very stable, and has not changed even a bit in the last two years.

There is a small piece however which is somehow bootstrapped. The lexer that recognizes the various elements in a `Compound` is generated using `lexyt`, given the lexical specification given in page 807.

Figures 7.1 on page 284 and 7.2 on page 285 show where this persistent storage format is used in the global `DURA` process.

⟨*Definition* ⟩≡
```
DEFINITION MODULE DuraLisp;

FROM ArrayList IMPORT ArrayList;
FROM Streams IMPORT OutputStream;

   ⟨Component class definition 797a⟩
   ⟨Atom class definition 798b⟩
   ⟨Compound class definition 801⟩

END DuraLisp;
```

⟨*Implementation* ⟩≡
```
IMPLEMENTATION MODULE DuraLisp;

FROM Conversions IMPORT IntConversions;
IMPORT InteLex;
FROM Streams IMPORT StdOut, StdErr, OutputStream;
IMPORT SYSTEM;

   ⟨Component class implementation 797b⟩
   ⟨Atom class implementation 799a⟩
   ⟨Compound class implementation 802a⟩

END DuraLisp;
```

Figure 7.15 Inheritance tree for `DURA`'s persistent Lisp-like format

```
                      ┌─ Atom (798)
Component (797) ──────┤
                      └─ Compound (801)
```

797a ⟨`Component` *class definition* 797a⟩≡

```
CLASS Component;
  METHOD AtomCode: INTEGER;
  METHOD CastToAtom: Atom;
  METHOD CastToCompound: Compound;
  METHOD Dump(Output: OutputStream;
              IndentLevel: INTEGER);
END Component;
```

797b ⟨`Component` *class implementation* 797b⟩≡

```
CLASS Component;

  ⟨Default Dump method 797c⟩
  ⟨Type casting methods 798a⟩

  DEFERRED METHOD SaveOnStream (Output: OutputStream);

END Component;
```

`Dump` is meant to be used for debugging purposes only. The default version of `Dump` as depicted here just outputs the white spaces required by the indentation level. This service is meant to be called through `BASE` from the derived versions of this method.

797c ⟨*Default* `Dump` *method* 797c⟩≡

```
METHOD Dump(Output: OutputStream;
            IndentLevel: INTEGER);
  BEGIN
  FOR i ∈ 0 .. IndentLevel - 1 DO
    Output.WriteChar (“ ”);
    END;
  END Dump;
```

`CastToCompound` and `CastToAtom` are convenience methods that are aimed at providing a more concise than explicit `WHAT` statement way of performing type tests. `CastToCompound` (resp. `CastToAtom`) is meant to be redefined by `Compound` (resp. `Atom`) and are defined here with a body made of an assertion rather than deferred methods to prevent the Y^A^F~L~ global optimizer [23] from complaining about calling a method that might not be defined.

798a ⟨*Type casting methods* 798a⟩≡

```
METHOD CastToCompound: Compound;
  BEGIN
  ASSERT FALSE;
  END CastToCompound;

METHOD CastToAtom: Atom;
  BEGIN
  ASSERT FALSE;
  END CastToAtom;

METHOD AtomCode: INTEGER;
  BEGIN
  RESULT ← CastToAtom.Code;
  END AtomCode;
```

`Atom` is the simplest form of `Component`. It can be of one of two kinds:

- It can have an integer `Code`, which must refer to one of the various lexemes defined in the lexical specification (See page 807)
- It can hold a text, as a character string.

This dichotomy might have been better implemented as two separate classes, but this is all simple enough to make such a move not worth the trouble.

798b ⟨`Atom` *class definition* 798b⟩≡

```
CLASS Atom;
  INHERITS Component;

  REDEFINE METHOD CREATE (Text: ARRAY OF CHAR;
                          Code: INTEGER);
  METHOD Text: ARRAY OF CHAR;
  METHOD IntegerValue: INTEGER;
  METHOD Code: INTEGER;
END Atom;
```

Only one of the two attributes `TheText` and `TheCode` listed here is considered as holding a significant value. `TheCode` will only be considered if `TheText` is `VOID`, thereby implementing the dichotomy among the two kinds of `Atom`s.

799a ⟨`Atom` *class implementation* 799a⟩≡
```
CLASS Atom;
  INHERITS Component;

  VAR
    TheText: ARRAY OF CHAR;
    TheCode: INTEGER;

  REDEFINE METHOD CREATE (Text: ARRAY OF CHAR;
                          Code: INTEGER);
    BEGIN
    BASE;
    TheText ← Text;
    TheCode ← Code;
    END CREATE;

  REDEFINE METHOD Dump(Output: OutputStream;
                       IndentLevel: INTEGER);
    BEGIN
    BASE (Output, IndentLevel);
    Output.WriteLine (TheText);
    END Dump;

  ⟨Trivial consultation methods for Atom, definition 799b⟩
  ⟨Redefining CastToAtom 800b⟩
  ⟨Saving an Atom 800c⟩

END Atom;
```

799b ⟨*Trivial consultation methods for* `Atom`*, definition* 799b⟩≡
```
METHOD Code: INTEGER;
  BEGIN
  RESULT ← TheCode;
  END Code;

METHOD Text: ARRAY OF CHAR;
  BEGIN
  RESULT ← TheText;
  END Text;
```

`IntegerValue` is a convenience method that performs a string to integer conversion on the text attached to the `Atom` denoted by `THIS`.

800a ⟨*Trivial consultation methods for* `Atom`*, definition* 799b⟩+≡
```
METHOD IntegerValue: INTEGER;
  BEGIN
  RESULT ← IntConversions.StringToInt (TheText);
  END IntegerValue;
```

We must redefine `CastToAtom` (See page 798) in order to return `THIS`. If applied to a `Compound`, `CastToAtom` should produce an error message.

800b ⟨*Redefining* `CastToAtom` 800b⟩≡
```
REDEFINE METHOD CastToAtom: Atom;
  BEGIN
  RESULT ← THIS;
  END CastToAtom;
```

The dual nature of `Atom`s as described above must be taken into account when saving them. If they have an attached text, it is simply written onto the file. Otherwise, a text representation of the token which would match their code is written instead.

800c ⟨*Saving an* `Atom` 800c⟩≡
```
REDEFINE METHOD SaveOnStream (Output: OutputStream);
  VAR
    TokenTable: ONCE ARRAY OF ARRAY OF CHAR;
    TheAtomCount: ONCE INTEGER;
  BEGIN
  ⟨Newline if required 800d⟩
  IF TheText ≠ Λ THEN
    Output.WriteString (TheText);
   ELSE
    IF TokenTable = Λ THEN
      ⟨Initialize TokenTable 807⟩
      END;
    ASSERT TokenTable_Code ≠ Λ;
    Output.WriteString (TokenTable_Code);
    END;
  END SaveOnStream;
```

In order to make the persistent files slightly easier to read, we insert a newline every 5 `Atom`.

800d ⟨*Newline if required* 800d⟩≡
```
IF TheAtomCount > 5 THEN
  Output.WriteLn;
  TheAtomCount ← 0;
  END;
TheAtomCount ← TheAtomCount + 1;
```

A Compound is basically a wrapper class around a list of Component, which, in turn, might also be Compounds, etc . . . It basically provides two sets of facilities:

- Providing a list-like interface (size, appending, extracting an element)
- Saving onto and retrieving from a file.

801 ⟨Compound *class definition* 801⟩≡

```
CLASS Compound;
  INHERITS Component;

  METHOD Elements: ArrayList(Component);

  METHOD Append (c: Component);
  METHOD AppendAtom (Code: INTEGER);
  METHOD AppendInteger (Value: INTEGER);
  METHOD AppendAtomText (Code: INTEGER;
                         Text: ARRAY OF CHAR);

  METHOD Size: INTEGER;
  METHOD Get(i: INTEGER): Component;
  REDEFINE METHOD CREATE;
  METHOD Read (FName: ARRAY OF CHAR): BOOLEAN;
  METHOD Save (FName: ARRAY OF CHAR);

END Compound;
```

802a ⟨Compound *class implementation* 802a⟩≡

```
CLASS Compound;
  INHERITS Component;

    VAR
      TheElements: ArrayList(Component);

  ⟨Redefining CastToCompound 802b⟩

  METHOD Elements: ArrayList(Component);
    BEGIN
    RESULT ← TheElements;
    END Elements;

  ⟨Appending Components 803a⟩

  METHOD Size: INTEGER;
    BEGIN
    RESULT ← TheElements.Size;
    END Size;

  METHOD Get(i: INTEGER): Component;
    BEGIN
    RESULT ← TheElements_i;
    END Get;

  REDEFINE METHOD CREATE;
    BEGIN
    BASE;
    TheElements.CREATE;
    END CREATE;

  ⟨Dump, as redefined by Compound 803b⟩
  ⟨Reading a Compound 804⟩
  ⟨Saving a Compound 806a⟩

END Compound;
```

We redefine `CastToCompound` (See page 798) in order to return `THIS`. If applied to a `Atom`, `CastToAtom` should produce a runtime error message.

802b ⟨*Redefining* CastToCompound 802b⟩≡

```
REDEFINE METHOD CastToCompound: Compound;
  BEGIN
  RESULT ← THIS;
  END CastToCompound;
```

The following methods are variations around a single theme: adding something to a `Compound`: adding a component, adding an atom given its code, adding an item given an integer value that must be stored in the atom's text attribute, etc . . .

803a ⟨*Appending* `Components` 803a⟩≡

```
METHOD Append (c: Component);
  BEGIN
  ASSERT c ≠ Λ;
  TheElements.Append (c);
  END Append;

METHOD AppendAtom (Code: INTEGER);
  BEGIN
  AppendAtomText (Code,
                  Text ← Λ);
  END AppendAtom;

METHOD AppendInteger (Value: INTEGER);
  BEGIN
  AppendAtomText (Code ← 0,
       Text ← IntConversions.IntToString (Value, 0));
  END AppendInteger;

METHOD AppendAtomText (Code: INTEGER;
                       Text: ARRAY OF CHAR);
  VAR
    c: Atom;
  BEGIN
  c.CREATE (Text, Code);
  Append (c);
  END AppendAtomText;
```

`Dump` basically outputs its elements, enclosed by parentheses.

803b ⟨`Dump`, *as redefined by* `Compound` 803b⟩≡

```
REDEFINE METHOD Dump(Output: OutputStream;
                     IndentLevel: INTEGER);
  BEGIN
  BASE (Output, IndentLevel);
  Output.WriteLine (“(”);
  FOR El ∈ TheElements DO
    El.Dump (Output, IndentLevel + 2);
    END;
  BASE (Output, IndentLevel);
  Output.WriteLine (“)”);
  END Dump;
```

Reading a `Compound` is based on a lexical analyzer `TheLex` of the class `InteLex` which is generated automatically based on the lexical specification described on page 807.

804 ⟨*Reading a* `Compound` 804⟩≡

```
METHOD Read(FName: ARRAY OF CHAR): BOOLEAN;
  VAR
    TheLex: InteLex;
    Tok: INTEGER;

  METHOD GetToken;
    BEGIN
    TheLex.Advance;
    Tok ← TheLex.CurrentToken;
    END GetToken;

  ⟨AcceptCompound local method 805b⟩
  ⟨AcceptComponent local method 805a⟩

  BEGIN
  TheLex.CREATE;
  RESULT ← TheLex.AttachFile (FName);
  IF RESULT THEN
    GetToken;
    ASSERT Tok = TheLex.BeginList;
    GetToken;
    TheElements ← AcceptCompound.TheElements;
    END;
  END Read;
```

`AcceptComponent` merely distinguishes between recognizing the begin of a list in which case `AcceptCompound` is called, and any other token, in which case a corresponding `Atom` is created.

805a ⟨`AcceptComponent` *local method* 805a⟩≡

```
METHOD AcceptComponent: Component;
  VAR
    TheAtom: Atom;
  BEGIN
  IF Tok = TheLex.BeginList THEN
    GetToken;
    RESULT ← AcceptCompound;
   ELSE
    TheAtom.CREATE (Text ← TheLex.Image,
                    Code ← TheLex.CurrentToken);
    RESULT ← TheAtom;
    GetToken;
    END;
  END AcceptComponent;
```

Not much to say . . .

805b ⟨`AcceptCompound` *local method* 805b⟩≡

```
METHOD AcceptCompound: Compound;
  BEGIN
  RESULT.CREATE;
  IF Tok ≠ TheLex.EndList THEN
    RESULT.Append (AcceptComponent);
    WHILE Tok = TheLex.Comma DO
      GetToken;
      RESULT.Append (AcceptComponent);
      END;
    END;
  GetToken;
  END AcceptCompound;
```

Given what `Read` looks like, there is little to say about `Save`. It is based on a local method named `SaveComponent` which, depending on the class of the component to save, calls `SaveAtom` or `SaveCompound`, which, in turn, calls `SaveComponent` recursively.

806a ⟨*Saving a* Compound 806a⟩≡

```
METHOD Save (FName: ARRAY OF CHAR);
  VAR
    TheOutput: OutputStream;
  BEGIN
  TheOutput.CREATE;
  TheOutput.Create (FName, TheOutput.WriteAccess);
  SaveOnStream (TheOutput);
  TheOutput.Close;
  END Save;
```

The `'['` and `','` are defined as special codes in the lexical specification in page 807.

806b ⟨*Saving a* Compound 806a⟩+≡

```
REDEFINE METHOD SaveOnStream (Output: OutputStream);
  BEGIN
  Output.WriteChar (“[”);
  FOR i ∈ 0 .. Size - 1 DO
    IF i > 0 THEN
      Output.WriteChar (“,”);
      END;
    THIS_i.SaveOnStream (Output);
    END;
  Output.WriteChar (“]”);
  END SaveOnStream;
```

`TokenTable` is a table that provides us with the ability to go from a code returned by a lexer matching the lexical specification given in page 807 back to the character string that has returned that code.

807 ⟨*Initialize* `TokenTable` 807⟩≡
```
TokenTable.CREATE (InteLex.LASTTOKEN+1);
TokenTable[InteLex.Case             ] ← “'CASE”;
TokenTable[InteLex.CaseAlt          ] ← “'CASEALT”;
TokenTable[InteLex.Concat           ] ← “'CONCAT”;
TokenTable[InteLex.Or               ] ← “'OR”;
TokenTable[InteLex.Repeat1          ] ← “'REPEAT1”;
TokenTable[InteLex.Repeat0          ] ← “'REPEAT0”;
TokenTable[InteLex.Optional         ] ← “'OPT”;
TokenTable[InteLex.Assign           ] ← “'ASSIGN”;
TokenTable[InteLex.Module           ] ← “'MODULE”;
TokenTable[InteLex.GrammarClause    ] ← “'CLAUSE”;
TokenTable[InteLex.GrammarRule      ] ← “'RULE”;
TokenTable[InteLex.Range            ] ← “'RANGE”;
TokenTable[InteLex.Terminal         ] ← “'TERM”;
TokenTable[InteLex.TerminalSet      ] ← “'TERMINALSET”;
TokenTable[InteLex.NonTerminal      ] ← “'NONTERM”;
TokenTable[InteLex.Attribute        ] ← “'ATTR”;
TokenTable[InteLex.IntegerType      ] ← “'INT”;
TokenTable[InteLex.BooleanType      ] ← “'BOOL”;
TokenTable[InteLex.StringType       ] ← “'STRING”;
TokenTable[InteLex.ListType         ] ← “'LIST”;
TokenTable[InteLex.This             ] ← “'THIS”;
```

The following chunk implements a lexical specification as recognized by `lexyt`, and which is used to produce a lexer to retrieve informations stored when saving (possibly nested) `Compoundss`. The lexer is generated under the name `InteLex`.

808 ⟨*durainte.yl* 808⟩≡

```
$BACKTRACKLEVEL 0

Ident          a-zA-Z[a-zA-Z0-9_]*
QualId        {Ident}\.{Ident}
Comma          \,
Case           \'CASE
CaseAlt        \'CASEALT
Concat         \'CONCAT
Or             \'OR
Repeat1        \'REPEAT1
Repeat0        \'REPEAT0
Optional       \'OPT
Assign         \'ASSIGN
BeginList      \[
EndList        \]
Module           \'MODULE
GrammarClause    \'CLAUSE
GrammarRule      \'RULE

Range            \'RANGE
Terminal         \'TERM
TerminalSet      \'TERMINALSET
NonTerminal      \'NONTERM


Attribute        \'ATTR
Integer        (\-|@)[0-9]+
IntegerType      \'INT
BooleanType      \'BOOL
ListType         \'LIST
StringType       \'STRING
This             \'THIS

NoRecovery              \'NORECOVERY
AutomaticRecovery       \'AUTOMATICRECOVERY

_Comment       \<[^>]*>
_Ws            [ \n\t]
```

7.24 DuraLookAhead

In an ideal OO compiler, syntactical analysis would merely be a question of creating nodes of the parse tree, and ask them to parse themselves based on a stream of tokens. In practice, this ideal view is not applicable: before creating the node, we must look at the stream of tokens to decide which kind of node comes next. We need some form of lookahead. YAFL

The plain YAFL compiler implements this lookahead service in a `LookAhead` class of which `StreamLookAhead` (used here as a base class for `DuraStreamLookAhead`) is a derived class. Creators (See method *DuraCreators.DuraStreamCreator. CreateStreamLookAhead* on page 406) ensure that whenever the YAFL compiler needs a `StreamLookAhead`, a `DuraStreamLookAhead` is created instead.

⟨*Definition* ⟩≡

```
DEFINITION MODULE DuraLookAhead;

  ⟨DuraLookAhead definition import list 816d⟩

  CLASS DuraStreamLookAhead;
    INHERITS StreamLookAhead;

    ⟨AcceptGrammarClause method definition 810a⟩
    ⟨AcceptGrammarRule method definition 810c⟩
    ⟨AcceptGrammarExpression method definition 812a⟩
    ⟨AcceptDuraIdent method definition 811b⟩

    ⟨AcceptClassDefinition, as redefined for DURA , definition 815b⟩
    ⟨AcceptClassImplementation, as redefined for DURA , definition 816b⟩

  END DuraStreamLookAhead;

END DuraLookAhead;
```

⟨*Implementation* ⟩≡

```
IMPLEMENTATION MODULE DuraLookAhead;

⟨DuraLookAhead implementation import list 816e⟩

   CLASS DuraStreamLookAhead;
     INHERITS StreamLookAhead;

     ⟨AcceptDeclaration, as redefined for DURA 815a⟩
     ⟨AcceptGrammarClause method implementation 810b⟩
     ⟨Accepting a grammar expression 811d⟩

     ⟨AcceptDuraIdent method implementation 811c⟩
     ⟨AcceptGrammarRule method implementation 811a⟩

     ⟨AcceptClassDefinition, as redefined for DURA , implementation 816a⟩
     ⟨AcceptClassImplementation, as redefined for DURA , implementation 816c⟩

   END DuraStreamLookAhead;

END DuraLookAhead;
```

AcceptGrammarClause (as well as AcceptGrammarRule and Accept-DuraIdent below) are ideal parsing methods according to the explanation given when starting this section: they just CREATE an instance of the class they must recognize, and call Parse to let the object actually parse itself. This implies of course that the lookahead symbol has been tested externally before calling any of these methods (See method *DuraClImplementation.DuraClassImplementation.ParseDeclarationList* on page 438).

810a ⟨AcceptGrammarClause *method definition* 810a⟩≡

```
METHOD AcceptGrammarClause: GrammarClause;
```

(See method *DuraGrClause.GrammarClause.Parse* on page 667)

810b ⟨AcceptGrammarClause *method implementation* 810b⟩≡

```
METHOD AcceptGrammarClause: GrammarClause;
   BEGIN
   RESULT.CREATE (LineNr, ColNr);
   RESULT.Parse (THIS);
   END AcceptGrammarClause;
```

810c ⟨AcceptGrammarRule *method definition* 810c⟩≡

```
METHOD AcceptGrammarRule: GrammarRule;
```

(See method *DuraGrClause.GrammarRule.Parse* on page 671)

811a ⟨`AcceptGrammarRule` *method implementation* 811a⟩≡

```
METHOD AcceptGrammarRule: GrammarRule;
  BEGIN
  RESULT.CREATE (LineNr, ColNr);
  RESULT.Parse (THIS);
  END AcceptGrammarRule;
```

811b ⟨`AcceptDuraIdent` *method definition* 811b⟩≡

```
METHOD AcceptDuraIdent: DuraIdent;
```

(See method *DuraIdentifiers.DuraIdent.Parse* on page 675)

811c ⟨`AcceptDuraIdent` *method implementation* 811c⟩≡

```
METHOD AcceptDuraIdent: DuraIdent;
  BEGIN
  RESULT.CREATE (LineNr, ColNr);
  RESULT.Parse (THIS);
  END AcceptDuraIdent;
```

After having set the foundations, we can now concentrate on actually recognizing grammar expressions. The following methods implement the grammar for grammar expressions. It is divided in the common $Expression$-$Term$-$Factor$ levels, to cope with operators with two levels of precedence (here: OR and implicit concatenation).

More formally, we have:

$$
\begin{array}{lcc}
GrammarExpression & ::= & GrammarExpression\ OR\ GrammarTerm\ | \\
 & & GrammarTerm \\
GrammarTerm & ::= & GrammarTerm\ GrammarFactor\ | \\
 & & GrammarFactor \\
GrammarFactor & ::= & GrammarFactor\ "+"\ | \\
 & & GrammarFactor\ "*"\ | \\
 & & GrammarFactor\ "?"\ | \\
 & & "["\ TerminalSetExpression\ "]"\ | \\
 & & Ident\ | \\
 & & Ident\ ":="\ GrammarFactor\ | \\
 & & CaseGrammarExpression\ | \\
 & & "("\ GrammarExpression\ ")"
\end{array}
$$

The grammar as described above is parsed using recursive descent as does the rest of the YAFL compiler, except for the fact that iteration (in $GrammarExpression$ and $GrammarTerm$) is represented using lists of arguments (See class *DuraExpressions. AggregateExpression* on page 449) instead of left or right recursion.

811d ⟨*Accepting a grammar expression* 811d⟩≡

⟨`AcceptGrammarExpression` *method implementation* 812b⟩
⟨`AcceptGrammarTerm` *method implementation* 812c⟩
⟨`AcceptGrammarFactor` *method implementation* 813⟩

812a ⟨AcceptGrammarExpression *method definition* 812a⟩≡

```
METHOD AcceptGrammarExpression: GrammarExpression;
```

812b ⟨AcceptGrammarExpression *method implementation* 812b⟩≡

```
METHOD AcceptGrammarExpression: GrammarExpression;
  VAR
    AggregateExpr: AggregateExpression;
    St: INTEGER;
  BEGIN
  RESULT ← AcceptGrammarTerm;
  St ← CurrentToken;
  WHILE (St = LexicalAnalyzer.Or) ∧ Ok DO
    IF AggregateExpr = Λ THEN
      AggregateExpr.CREATE (LineNr, ColNr, St);
      AggregateExpr.Append (RESULT);
      RESULT ← AggregateExpr;
      END;
    GetToken;
    AggregateExpr.Append (AcceptGrammarTerm);
    St ← CurrentToken;
    END;
  END AcceptGrammarExpression;
```

812c ⟨AcceptGrammarTerm *method implementation* 812c⟩≡

```
METHOD AcceptGrammarTerm: GrammarExpression;
  VAR
    St: INTEGER;
    TheAggregateExpr: AggregateExpression;
  BEGIN
  RESULT ← AcceptGrammarFactor;
  ASSERT Ok ⇒ RESULT ≠ Λ;
  St ← CurrentToken;
  WHILE Ok ∧ ((St = LexicalAnalyzer.Ident) ∨
              (St = LexicalAnalyzer.LeftParen) ∨
              (St = LexicalAnalyzer.Case)) DO
    IF TheAggregateExpr = Λ THEN
      TheAggregateExpr.CREATE (LineNr, ColNr,
                               LexicalAnalyzer.Dot);
      TheAggregateExpr.Append (RESULT);
      RESULT ← TheAggregateExpr;
      END;
    TheAggregateExpr.Append(AcceptGrammarFactor);
    St ← CurrentToken;
    END;
  END AcceptGrammarTerm;
```

813 ⟨AcceptGrammarFactor *method implementation* 813⟩≡

```
METHOD AcceptGrammarFactor: GrammarExpression;
  VAR
    TheAtom: AtomExpression;
    TheOptExpr: OptionalExpression;
    TheRepExpr: RepetitionExpression;
    TheCaseExpr: CaseExpression;
    TheTerminalSetExpr: TerminalSetExpression;
    St: INTEGER;
    Id: DuraIdent;
  BEGIN
  St ← CurrentToken;
  CASE St OF
    LexicalAnalyzer.LeftBracket:
      ⟨Accept a terminal set 814a⟩
      END;
    LexicalAnalyzer.LeftParen:
      GetToken;
      RESULT ← AcceptGrammarExpression;
      Accept (LexicalAnalyzer.RightParen);
      END;
    LexicalAnalyzer.Ident:
      Id ← AcceptDuraIdent;
      IF CurrentToken = LexicalAnalyzer.Becomes THEN
        GetToken;
        RESULT ← AcceptGrammarFactor;
        RESULT.SetTarget (Id);
       ELSE
        TheAtom.CREATE (LineNr, ColNr, Id);
        RESULT ← TheAtom;
        END;
      END;
    LexicalAnalyzer.Case:
      TheCaseExpr.CREATE (LineNr, ColNr);
      TheCaseExpr.Parse(THIS);
      RESULT ← TheCaseExpr;
      END;
   ELSE
    Error (“Grammar factor Expected”);
    END;
  ⟨Consider a trailing “+” or “*” 814b⟩
  END AcceptGrammarFactor;
```

The terminal set must be created, and can then parse itself (See method *DuraExpressions.TerminalSetExpression.Parse* on page 466).

814a ⟨*Accept a terminal set* 814a⟩≡
```
TheTerminalSetExpr.CREATE (LineNr, ColNr);
TheTerminalSetExpr.Parse(THIS);
RESULT ← TheTerminalSetExpr;
```

If the grammar factor is followed by a "+" or a "*", we build a `RepetitionExpression` (See page 453) in `TheRepExpr`, while we build an `OptionalExpression` (See page 451) if it is followed by a "?".

814b ⟨*Consider a trailing ''+" or ''*"* 814b⟩≡
```
St ← CurrentToken;
IF (St = LexicalAnalyzer.Star) ∨
    (St = LexicalAnalyzer.Plus) THEN
  TheRepExpr.CREATE (LineNr, ColNr, RESULT, St=LexicalAnalyzer.Plus);
  GetToken;
  IF CurrentToken = LexicalAnalyzer.By THEN
    GetToken;
    TheRepExpr.SetSeparator(AcceptGrammarFactor);
    END;
  RESULT ← TheRepExpr;
 ELSIF St = LexicalAnalyzer.QuestionMark THEN
  TheOptExpr.CREATE (LineNr, ColNr, RESULT);
  RESULT ← TheOptExpr;
  GetToken;
  END;
```

`AcceptDeclaration` is redefined here, to deal with the leading `NONTERMINAL` keyword `DURA` must be able to handle. The idea is very simple: if the leading token is `NONTERMINAL`, this redefined version will call the methods `AcceptClassImplementation` or `AcceptClassDefinition` depending on the context because we are aware that a class declaration in `DURA` can start with this keyword, which holds no meaning for the plain YAFL compiler. In all other cases, that is, when the leading token is anything else, we call `BASE` to fall back on the original behaviour of this method.

815a ⟨`AcceptDeclaration`, *as redefined for* `DURA` 815a⟩≡

```
REDEFINE METHOD AcceptDeclaration
                         (Implementation: BOOLEAN): DeclList;
  VAR
    TheCode: INTEGER;
    TheList: DeclList (Declaration);
  BEGIN
  TheList.CREATE;
  RESULT ← TheList;
  IF CurrentToken = UdfToken THEN
    TheCode ← CurrentUdfCode;
    IF TheCode = DuraLex.NonTerminal THEN
      IF Implementation THEN
        TheList.Append (AcceptClassImplementation);
       ELSE
        TheList.Append (AcceptClassDefinition);
        END;
      END;
   ELSE
    RESULT ← BASE (Implementation);
    END;
  ASSERT RESULT ≠ Λ;
  END AcceptDeclaration;
```

`AcceptClassDefinition` (as well as `AcceptClassImplementation`, see below) as redefined here makes use of a special YAFL feature, named *constrained redefinition* [31]. The idea is rather simple: one can redefine a method as returning a type which is *derived* from the type returned by the method it redefines. This feature is sound typewise, and improves type checking considerably, since a more accurate type information is available. The only requirement is that such redefinition, when performed on a public method, must be public as well, that is, the redefinition must be present in the definition module as well. YAFL

815b ⟨`AcceptClassDefinition`, *as redefined for* `DURA` , *definition* 815b⟩≡

```
REDEFINE METHOD AcceptClassDefinition: DuraClassDefinition;
```

816a ⟨AcceptClassDefinition, *as redefined for* DURA , *implementation* 816a⟩≡
```
REDEFINE METHOD AcceptClassDefinition: DuraClassDefinition;
  BEGIN
  RESULT.CREATE (LineNr, ColNr);
  RESULT.Parse (THIS);
  END AcceptClassDefinition;
```

816b ⟨AcceptClassImplementation, *as redefined for* DURA , *definition* 816b⟩≡
```
REDEFINE METHOD AcceptClassImplementation: DuraClassImplementation;
```

816c ⟨AcceptClassImplementation, *as redefined for* DURA , *implementation* 816c⟩≡
```
REDEFINE METHOD AcceptClassImplementation: DuraClassImplementation;
  BEGIN
  RESULT.CREATE (LineNr, ColNr);
  RESULT.Parse (THIS);
  END AcceptClassImplementation;
```

816d ⟨DuraLookAhead *definition import list* 816d⟩≡
```
FROM StreamLookAhead IMPORT StreamLookAhead;
FROM YaflClDefinition IMPORT ClassDefinition;
FROM YaflClImplementation IMPORT ClassImplementation;
FROM YaflNTList IMPORT DeclList;
FROM YaflType IMPORT TypedNonTerminal;

FROM DuraClDefinition IMPORT DuraClassDefinition;
FROM DuraClImplementation IMPORT DuraClassImplementation;
FROM DuraGrClause IMPORT GrammarClause, GrammarRule;
FROM DuraIdentifiers IMPORT DuraIdent;
FROM DuraExpressions IMPORT GrammarExpression;
```

816e ⟨DuraLookAhead *implementation import list* 816e⟩≡
```
FROM YaflDeclarations IMPORT Declaration;
IMPORT DuraLex;
FROM YaflLex IMPORT LexicalAnalyzer;
FROM Streams IMPORT StdOut;

FROM DuraExpressions IMPORT GrammarExpression, AggregateExpression,
                            AtomExpression,
                            CaseExpression, CaseExpressionAlt,
                            OptionalExpression, TerminalSetExpression,
                            RepetitionExpression;
```

7.25 DuraModules

YAFL

This module implements the `DuraDefinitionModule` class, which is meant to replace the `DefinitionModule` class in YAFL parse trees by the virtue of creators (See method *DuraCreators.DuraObjectCreator.CreateDefinitionModule* on page 405). It provides a few simple `DURA`-specific services, such as retrieving the intermediate representation attached to a module (See method *DuraModules.DuraDefinitionModule.IntermediateModule* on the following page) or `DURA`-specific semantic analysis (See method *DuraModules.DuraDefinitionModule.DuraTag* on page 820).

⟨*Definition* ⟩≡
```
DEFINITION MODULE DuraModules;

⟨DuraModules definition import list 820c⟩

   ⟨DuraDefinitionModule class definition 817⟩

END DuraModules;
```

⟨*Implementation* ⟩≡
```
IMPLEMENTATION MODULE DuraModules;

⟨DuraModules implementation import list 820d⟩

⟨DuraDefinitionModule class implementation 818a⟩

END DuraModules;
```

817 ⟨`DuraDefinitionModule` *class definition* 817⟩≡
```
CLASS DuraDefinitionModule;
   INHERITS DefinitionModule;

   REDEFINE METHOD GetClass (ClassName: ARRAY OF CHAR):
                                           DuraClassDefinition;

   METHOD DuraClasses: DeclList (DuraClassDefinition);
   METHOD IntermediateModule: InterModule;

   ⟨DuraTag method definition 820a⟩

END DuraDefinitionModule;
```

818a ⟨DuraDefinitionModule *class implementation* 818a⟩≡

```
CLASS DuraDefinitionModule;
  INHERITS DefinitionModule;

  ⟨InterModuleName method 818c⟩

  ⟨IntermediateModule method implementation 818b⟩
  ⟨GetClass, as redefined by DuraDefinitionModule 819a⟩

  ⟨DuraClasses method implementation 819b⟩
  ⟨DuraTag method implementation 820b⟩

END DuraDefinitionModule;
```

IntermediateModule returns the intermediate representation for the grammatical information attached to a module (See class *DuraIntermediateFormat.InterModule* on page 754). IntermediateModule assumes that this information has been produced in some earlier phase, and reads the information back from a disk file (See method *DuraIntermediateFormat.InterModule.ParseFile* on page 756).

818b ⟨IntermediateModule *method implementation* 818b⟩≡

```
VAR
  TheInter: InterModule;

METHOD IntermediateModule: InterModule;
  BEGIN
  IF TheInter = Λ THEN
    TheInter.CREATE;
    IF ¬ TheInter.ParseFile (InterModuleName) THEN
      TheInter ← Λ;
      StdErr.WriteLine (‘‘Cannot parse file: ” + InterModuleName +
                        ‘‘ (” + Id.Data + ‘‘)”);
      END;
    END;
  RESULT ← TheInter;
  END IntermediateModule;
```

YAFL

InterModuleName builds the file name for the intermediate “.cy” file attached to a module (See figure 7.1 in page 284). It uses a facility provided by the plain YAFL compiler to build a file name for intermediate compilation files, given a radix and an extension.

818c ⟨InterModuleName *method* 818c⟩≡

```
METHOD InterModuleName: ARRAY OF CHAR;
  BEGIN
  RESULT ← GetWorld.BuildTargetFName (Id.Data, DuraCfg.DuraExtension);
  END InterModuleName;
```

`GetClass`, as redefined here, serve a single purpose: improve typing by taking advantage of the fact that whenever the Y^A F_L compiler will need a `ClassDefinition`, a `DuraClassDefinition` will be created instead (See method *DuraLookAhead. DuraStreamLookAhead.AcceptClassDefinition* on page 815). Hence, instead of having `GetClass` return a `ClassDefinition`, it is perfectly safe to redeclare it as returning a `DuraClassDefinition`. The body of this method does nothing more than perform the actual type test.

819a ⟨`GetClass`, *as redefined by* `DuraDefinitionModule` 819a⟩≡

```
REDEFINE METHOD GetClass (ClassName: ARRAY OF CHAR): DuraClassDefinition;
  VAR
    Cl: ClassDeclaration;
  BEGIN
  Cl ← BASE(ClassName);
  IF Cl ≠ Λ THEN
    WHAT Cl OF
      IN DuraClassDefinition:
        RESULT ← TAG;
        END;
      END;
    END;
  END GetClass;
```

`DuraClasses` returns the list of all non-terminal classes (See method *DuraClDefinition.DuraClassDefinition.IsNonTerminal* on page 418) in the module, retaining the result in an attribute for efficient further retrieval.

819b ⟨`DuraClasses` *method implementation* 819b⟩≡

```
VAR
  TheDuraClasses: DeclList (DuraClassDefinition);

METHOD DuraClasses: DeclList (DuraClassDefinition);
  BEGIN
  IF TheDuraClasses = Λ THEN
    TheDuraClasses.CREATE;
    FOR Cl ∈ Classes DO
      WHAT Cl OF
        IN DuraClassDefinition:
          IF TAG.IsNonTerminal THEN
            TheDuraClasses.Append (TAG);
            END;
          END;
        END;
      END;
    END;
  RESULT ← TheDuraClasses;
  END DuraClasses;
```

820a ⟨DuraTag *method definition* 820a⟩≡

```
METHOD DuraTag (Spec: SyntacticSpecification);
```

DuraTag is a DURA-specific semantic analysis method, which happens to be rather simple, since it invokes DuraTag as defined for DuraClassDefinition (See page 419).

820b ⟨DuraTag *method implementation* 820b⟩≡

```
METHOD DuraTag (Spec: SyntacticSpecification);
BEGIN
CurrentSpot.PushCurrentModule (THIS);
FOR Cl ∈ DuraClasses DO
  Cl.DuraTag(Spec);
  END;
CurrentSpot.PopCurrentModule;
END DuraTag;
```

820c ⟨DuraModules *definition import list* 820c⟩≡

```
FROM DuraClDefinition IMPORT DuraClassDefinition;
FROM DuraIntermediateFormat IMPORT InterModule, InterClause;
FROM DuraSpecification IMPORT SyntacticSpecification;

FROM YaflModules IMPORT DefinitionModule;
FROM YaflNTList IMPORT DeclList;
```

820d ⟨DuraModules *implementation import list* 820d⟩≡

```
FROM DuraLisp IMPORT Compound;
IMPORT DuraCfg;
IMPORT InteLex;
FROM DuraIntermediateFormat IMPORT InterModule;
IMPORT Space;
FROM Streams IMPORT StdOut, StdErr;

FROM YaflCfg IMPORT CurrentSpot;
FROM YaflClasses IMPORT ClassDeclaration;
FROM YaflModules IMPORT DefinitionModule;
```

7.26 DuraReductions

This module describes the various forms of *reductions* that are supported by DURA. A reduction is an action that can be taken when a grammar rule (or more accurately, a grammar rule's right hand side (See class *DuraFlat.FlatRHS* on page 513)) has been recognized entirely.

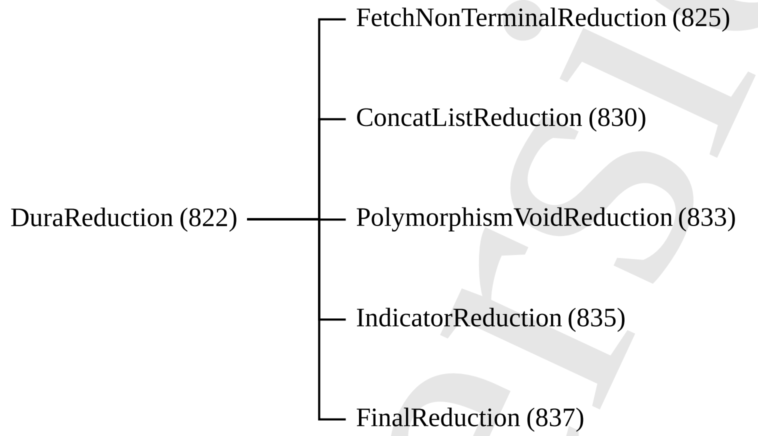
In most yacc-like tools, there is a single form of reduction: call a user-defined action that will do something with the top of the stack, remove the proper number of items from the stack, and replace them with the resulting item of the user-defined code.

In DURA, things are slightly different. Depending on the nature of the grammar rule, different actions can be taken. For instance, consider the following case. We have a Statement nonterminal class, and IfStatement inherits from Statement. DURA will translate this inheritance relationship into a grammar rule $Statement ::= IfStatement$. Amazingly enough, the reduction here simply does nothing. It will leave the IfStatement instance on the stack, and everything will be fine, since an IfStatement instance is indeed a valid Statement instance as well.

Similarly, the reductions that will have to be attached to grammar rules that have been generated automatically to implement repetition operators will need specific reduction action, to build the high-level – Understand: using strongly typed parameterized classes – representation of the linked list.

⟨*Definition* ⟩≡
```
DEFINITION MODULE DuraReductions;

FROM DuraClDefinition IMPORT DuraClassDefinition;
FROM DuraFlat IMPORT FlatRHS;
FROM DuraIntermediateFormat IMPORT InterRule,
                                   InterExpression,
                                   InterAttribute;
FROM DuraOutput IMPORT DuraOutputStream;
FROM Streams IMPORT OutputStream;

  ⟨DuraReduction base class definition 822a⟩

  ⟨FetchNonTerminalReduction class definition 825c⟩
  ⟨ConcatListReduction class definition 830c⟩
  ⟨PolymorphismVoidReduction class definition 833⟩
  ⟨IndicatorReduction class definition 835b⟩
  ⟨FinalReduction class definition 837c⟩

END DuraReductions;
```

Figure 7.16 Inheritance tree for `DuraReduction`



⟨*Implementation* ⟩≡
```
IMPLEMENTATION MODULE DuraReductions;

FROM DuraIntermediateFormat IMPORT InterThis;
FROM DuraFlat IMPORT FlatNonTerminal;
FROM Streams IMPORT StdOut;

  ⟨DuraReduction class implementation 824b⟩
  ⟨ConcatListReduction class implementation 831⟩
  ⟨FetchNonTerminalReduction class implementation 826⟩
  ⟨PolymorphismVoidReduction class implementation 834⟩
  ⟨IndicatorReduction class implementation 836⟩
  ⟨FinalReduction class implementation 838a⟩

END DuraReductions;
```

7.26.1 `DuraReduction` base class

`DuraReduction` is the base class for all the reductions that can be performed by DURA.

822a ⟨`DuraReduction` *base class definition* 822a⟩≡
```
CLASS DuraReduction;
```

Its services include dumping onto an output stream for debugging purposes...

822b ⟨`DuraReduction` *base class definition* 822a⟩+≡
```
DEFERRED METHOD Dump (Output: OutputStream);
```

as well as an indicator whether the reduction is implicit, that is, whether something happens or not.

823a ⟨`DuraReduction` *base class definition* 822a⟩+≡

```
  METHOD Implicit: BOOLEAN;
```

The following two methods are related to the code generation process. The first one performs the reduce, that is, the extraction of the top slice of the LR stack, and its replacement by a `ReduceStackElement` (See page 337).

The second one generates code to actually fetch some information from a reduced slice onto objects. There are typically two forms of such fetches:

- One that fetches values into attributes, thereby actually building the parse tree
- One that builds a list when dealing with a repetition grammar rule.

823b ⟨`DuraReduction` *base class definition* 822a⟩+≡

```
    METHOD GenerateReductionCode (Output: DuraOutputStream;
                                  RHS: FlatRHS);
    METHOD GenerateFetchCode (Output: DuraOutputStream;
                              RHS: FlatRHS);
```

The last generic service provided by `DuraReduction` is the ability to be compared for equality. This will be used to check whether two flattened grammar rules are equal, when considering optimization by merging.

Quite obviously, this method will only return `TRUE` when `Other` is of the same class as `THIS`. This could be ensured by using covariance[11]:

```
%
CLASS DuraReduction(Cov IN DuraReduction);
...
  DEFERRED METHOD IsEqual(Other: Cov): BOOLEAN;
...
END DuraReduction;
```

and then, while defining a derived class, by indicating explicitely the class they can be compared with for equality:

```
%
CLASS FetchNonTerminalReduction
  INHERITS DuraReduction(FetchNonTerminalReduction);
...
  REDEFINE METHOD IsEqual(Other:
              FetchNonTerminalReduction): BOOLEAN;
...
END FetchNonTerminalReduction;
```

[11]In this context, covariance is a technique where one refines the parameter of parameterized classes together with the refinement of inheritance.

Within the so redefined version of `IsEqual`, `Other` is known as a `FetchNonTerminalReduction`, and can be compared easily for equality.

This feature is apparently better supported in Eiffel [86] using `like current` clauses, but this introduces a flaw in typing that has been shown in [87]. Covariance as supported by Y^AF_L is somehow less convenient (understand: more verbose) but is safe from a typing point of view.

Nevertheless, covariance will not be not used here. Rather than requiring this class to be parameterized just for this `IsEqual` method, we rather use the WHAT statement explicitely in each redefinition.

824a ⟨`DuraReduction` *base class definition* 822a⟩+≡

```
    DEFERRED METHOD IsEqual (Other: DuraReduction): BOOLEAN;
  END DuraReduction;
```

824b ⟨`DuraReduction` *class implementation* 824b⟩≡

```
  CLASS DuraReduction;

    ⟨GenerateReduceTopSlice method implementation 825a⟩
    ⟨Default GenerateReductionCode implementation 825b⟩

    METHOD GenerateFetchCode (Output: DuraOutputStream;
                              RHS: FlatRHS);
      BEGIN
      END GenerateFetchCode;
```

The default behaviour for `Implicit` is to return `FALSE`, but the derived classes can redefine this method to indicate in which cases the reduction they represent is not supposed to be implicit.

824c ⟨`DuraReduction` *class implementation* 824b⟩+≡

```
    METHOD Implicit: BOOLEAN;
      BEGIN
      RESULT ← FALSE;
      END Implicit;

  END DuraReduction;
```

`GenerateReduceTopSlice` generates code that reduces the top of the LR stack, by means of the `ReduceTopSlice` method (See page 877). The number of elements to reduce is deduced from the attached right-hand side, and the target state the parser must go to is computed by a call generated by `GenerateGotoMethodCall` (See page 501).

825a ⟨`GenerateReduceTopSlice` *method implementation* 825a⟩≡

```
  METHOD GenerateReduceTopSlice(Output: DuraOutputStream;
                              RHS: FlatRHS);
    BEGIN
    Output.WriteString (‘‘ReduceTopSlice (”);
    Output.WriteInt (RHS.Size, 0);
    Output.WriteString (‘‘,”);
    Output.WriteInt (RHS.Nr, 0);
    Output.WriteString (‘‘, ”);
    RHS.GetFlatRule.GenerateGotoMethodCall(Output, RHS.Size);
    Output.WriteLine (‘‘);”);
    END GenerateReduceTopSlice;
```

825b ⟨*Default* `GenerateReductionCode` *implementation* 825b⟩≡

```
  METHOD GenerateReductionCode (Output: DuraOutputStream;
                              RHS: FlatRHS);
    BEGIN
    Output.WriteLine (‘‘BEGIN”);
    Output.WriteString (‘‘VOID := ”);
    GenerateReduceTopSlice (Output, RHS);
    END GenerateReductionCode;
```

7.26.2 `FetchNonTerminalReduction`

`FetchNonTerminalReduction` is the class that describes a reduction where one or more attributes are assigned with parts of the reduction.

825c ⟨`FetchNonTerminalReduction` *class definition* 825c⟩≡

```
  CLASS FetchNonTerminalReduction;
    INHERITS DuraReduction;

    REDEFINE METHOD CREATE (Rule: InterRule);
    METHOD Rule: InterRule;

  END FetchNonTerminalReduction;
```

826 ⟨FetchNonTerminalReduction *class implementation* 826⟩≡

```
CLASS FetchNonTerminalReduction;
  INHERITS DuraReduction;

  VAR
    TheRule: InterRule;

  REDEFINE METHOD CREATE (Rule: InterRule);
    BEGIN
    TheRule ← Rule;
    BASE;
    END CREATE;

  METHOD Rule: InterRule;
    BEGIN
    RESULT ← TheRule;
    END Rule;

  REDEFINE METHOD Dump(Output: OutputStream);
    BEGIN
    Output.WriteString (“Fetch non-terminal: ” +
                        TheRule.EnclosingClause.ClassName + “(”);
    Output.WriteInt (TheRule.EnclosingClause.RuleList.Index (TheRule), 0);
    Output.WriteChar (“)”);
    END Dump;

  ⟨Checking wether two FetchNonTerminalReduction’s are equal 827⟩
  ⟨GenerateReductionCode, as redefined by FetchNonTerminalReduction 828a⟩

END FetchNonTerminalReduction;
```

`Other` is equivalent to `THIS` if:

- `Other` is also a `FetchNonTerminalReduction`, just as `THIS` is, unless they cannot be equal anyway.
- They both refer to the same intermediate grammar rule.

827 ⟨*Checking wether two* `FetchNonTerminalReduction`*'s are equal* 827⟩≡

```
REDEFINE METHOD IsEqual (Other: DuraReduction): BOOLEAN;
  BEGIN
  IF Other ≠ Λ THEN
    WHAT Other OF
      IN FetchNonTerminalReduction:
        RESULT ← Rule = TAG.Rule;
        END;
     ELSE
      END;
    END;
  END IsEqual;
```

A `FetchNonTerminalReduction` cannot be considered as implicit, in the sense that even in cases where it does little, it must be retained. It cannot be removed by some kind of optimization process. Otherwise, the grammar's semantics will not be represented faithfully.

For instance, when a grammar rule within a nonterminal includes a single reference to another nonterminal, *without any inheritance relationship*, the intermediate grammar rule cannot be eliminated:

```
NONTERMINAL CLASS PackagedStatement;
  INHERITS BaseNonTerminal;

  GRAMMAR
    TheStatement;

  VAR
    TheStatement: Statement;
```

A reference to a `PackagedStatement` cannot be replaced by a reference to a plain `Statement`, since the resulting parse tree would be typewise incorrect.

Hence, we do not redefine the `Implicit` method (See page 823), and retain its default behaviour, which is to return `FALSE`.

GenerateReductionCode considers two cases:

- The rule includes an assignment to THIS (See page 271)
- The rule does not include such an assignment, and zero or more attributes must be fetched

828a ⟨*GenerateReductionCode, as redefined by* FetchNonTerminalReduction 828a⟩≡

```
REDEFINE METHOD GenerateReductionCode (Output: DuraOutputStream;
                                       RHS: FlatRHS);
  VAR
    Cl: DuraClassDefinition;
    ThisIndex: INTEGER;
  BEGIN
  Cl ← Rule.EnclosingClause.Class;
  ASSERT Cl ≠ Λ;
  ⟨Search for assignment to THIS, store index in ThisIndex 828b⟩
  IF ThisIndex ≥ 0 THEN
    ⟨Generate trivial code for assignment to THIS 828c⟩
   ELSE
    ⟨Generate less trivial code for the creation of a new non-terminal 829a⟩
    END;
  END GenerateReductionCode;
```

We make use of YAFL quantifiers [83] to detect whether RHS includes a reference to THIS.

828b ⟨*Search for assignment to* THIS*, store index in* ThisIndex 828b⟩≡

```
ThisIndex ← FIRST i ∈ 0 .. RHS.Size-1 • RHS_i.Target = InterThis;
```

If we deal with an assignment to THIS, we first reduce the right amount of LR stack space (See method *DuraReductions.DuraReduction.GenerateReduceTopSlice* on page 825), and then *transfer* the non-terminal in this topmost slice to the newly created reduction (See class *DuraBase.ReduceStackElement* on page 337).

828c ⟨*Generate trivial code for assignment to* THIS 828c⟩≡

```
Output.WriteLine (“VAR”);
Output.IncIndent (2);
Output.WriteLine (“Top: ReduceStackElement;”);
Output.DecIndent (2);
Output.WriteLine (“BEGIN”);
Output.WriteString (“Top := ”);
GenerateReduceTopSlice(Output, RHS);
Output.WriteString(“Top.SetNonTerminal (ExtractNonTerminal(Top.Array[”);
Output.WriteInt (ThisIndex, 0);
Output.WriteLine(“]));”);
```

Otherwise, we generate code that `CREATE`s a new instance of the class that encloses the RHS we try to reduce (See method *DuraClDefinition.DuraClassDefinition.GenerateCreationMethodName* on page 431), reduce the topmost slice as explained in the previous chunk, and fetch attributes from the `ReduceStackElement Top` by generating a call to `FetchAttributes` (See page 884).

829a ⟨*Generate less trivial code for the creation of a new non-terminal* 829a⟩≡

```
Output.WriteLine (“VAR”);
Output.IncIndent (2);
Output.WriteLine (“p: ” + Cl.YaflClassName + “;”);
Output.WriteLine (“Top: ReduceStackElement;”);
Output.DecIndent (2);
Output.WriteLine (“BEGIN”);
Output.WriteString (“p := ”);
Cl.GenerateCreationMethodName(Output);
Output.WriteLine (“;”);
Output.WriteString (“Top := ”);
GenerateReduceTopSlice(Output, RHS);
Output.WriteLine (“Top.SetNonTerminal (p);”);
Output.WriteLine (“FetchAttributes(Top, p);”);
```
YAFL

When a new non-terminal is created by the parser, it is attached to the corresponding metaclass (See method *DuraClDefinition.DuraClassDefinition.GenerateMetaClassVariable* on page 428), as the default instance. After the fetch of all attributes for this instance, we generate code that removes the association between the non-terminal and the corresponding metaclass so that illegal fetches (that is, fetches into a non-terminal which is not currently being reduced) are easier to detect. For instance, when fetching attributes for a `FunctionCall` non-terminal in a context where no `FunctionCall` is being recognized – for instance, because of some malfunction of the parser generation process – the fact that no current instance is attached to the metaclass for `FunctionCalls` will cause the parser to crash instantly.

Besides, this also reduces the unnecessary links between the meta-class system and the parse tree that might cause the garbage collector to retain objects which serve no purpose any longer. Automatic garbage collection is of course much simpler than manual memory management, but one must make sure that no dangling reference prevents the space for useless data from being reclaimed. YAFL

829b ⟨*Generate less trivial code for the creation of a new non-terminal* 829a⟩+≡

```
Cl.GenerateMetaClassVariable (Output);
Output.WriteLine (“.UseInstance(VOID);”);
```

After the fetch, the generated non-terminal is asked whether if it is ok (See method *DuraBase.BaseNonTerminal.Commit* on page 347). If not, backtracking starts (See *Parser skeleton.Parser skeleton.TheFailure* on page 869).

830a ⟨*Generate less trivial code for the creation of a new non-terminal* 829a⟩+≡
```
Output.WriteLine (``IF NOT p.Commit THEN");
Output.IncIndent (2);
Output.WriteLine (``p.RollBack;");
Output.WriteLine (``TheFailure := TRUE;");
Output.Else;
```

If everything happened flawlessly, and if the class `Cl` has been marked as not supporting backtracking, code is generated to allow the deallocation of the data structures that are required for backtracking (See method *DuraBase.ReduceStackElement.Cut* on page 344).

830b ⟨*Generate less trivial code for the creation of a new non-terminal* 829a⟩+≡
```
IF Cl.CutLevel ≠ Cl.NoCut THEN
  Output.WriteString (``Top.Cut(");
  Output.WriteInt (Cl.CutLevel, 0);
  Output.WriteLine (``);");
  END;
Output.End;
```

7.26.3 ConcatListReduction

830c ⟨`ConcatListReduction` *class definition* 830c⟩≡
```
CLASS ConcatListReduction;
  INHERITS DuraReduction;

END ConcatListReduction;
```

`ConcatListReduction` implements the high-level iteration facility (the "+" and "*" operators) provided by `DURA` . A grammatical expression such as `A` + is translated in the flattened form:

$$\begin{array}{lll} Synth & ::= & Synth\ A\ | \\ & & A \end{array}$$

The `ConcatListReduction` is attached to the first `RHS` $Synth\ A$, and generates code that attaches the A at the tail of $Synth$.

831 ⟨`ConcatListReduction` *class implementation* 831⟩≡

```
CLASS ConcatListReduction;
  INHERITS DuraReduction;

  REDEFINE METHOD Dump(Output: OutputStream);
    BEGIN
    Output.WriteString (" cat list");
    END Dump;

  REDEFINE METHOD IsEqual (Other: DuraReduction): BOOLEAN;
    BEGIN
    IF Other ≠ Λ THEN
      WHAT Other OF
        IN ConcatListReduction:
          RESULT ← TRUE;
          END;
       ELSE
        END;
      END;
    END IsEqual;

  ⟨GenerateReductionCode, as redefined by ConcatListReduction 832⟩

END ConcatListReduction;
```

The code generation for `ConcatListReduction` connects (See method *DuraBase.BaseNonTerminal.SetPrevious* on page 356) the attached non-terminal of the rightmost element in the RHS to the non-terminal of the leftmost element in the RHS.

832 ⟨`GenerateReductionCode`, *as redefined by* `ConcatListReduction` 832⟩≡

```
  REDEFINE METHOD GenerateReductionCode (Output: DuraOutputStream;
                                         RHS: FlatRHS);
    BEGIN
    Output.WriteLine (‘‘VAR”);
    Output.IncIndent (2);
    Output.WriteLine (‘‘Top: ReduceStackElement;”);
    Output.WriteLine (‘‘Nt: BaseNonTerminal;”);
    Output.DecIndent (2);
    Output.WriteLine (‘‘BEGIN”);
    Output.WriteString (‘‘Top := ”);
    GenerateReduceTopSlice(Output, RHS);
    WHAT RHS.Last OF
      IN FlatNonTerminal:
        Output.WriteString(‘‘Nt := ExtractNonTerminal(Top.Array[”);
        Output.WriteInt (RHS.Size - 1, 0);
        Output.WriteLine(‘‘]);”);

        Output.WriteLine(‘‘IF Nt <> VOID THEN”);
        Output.IncIndent (2);
        Output.WriteLine (‘‘Top.SetNonTerminalNoConnect (Nt);”);
        Output.WriteLine(‘‘Nt.SetPrevious” +
                         ‘‘(ExtractNonTerminal(Top.Array[0]));”);
        Output.End;
        END;
     ELSE
      Output.WriteLine (‘‘-- No attachment when recognizing ” +
                        ‘‘lists of terminals”);
      END;
    END GenerateReductionCode;
```

7.26.4 PolymorphismVoidReduction

`PolymorphismVoidReduction` does little. It implements the reduction related to reverse inheritance, that is, the reduction of a rule of the form $A\ ::=\ B$ where B inherits from A.

833 ⟨PolymorphismVoidReduction *class definition* 833⟩≡

```
CLASS PolymorphismVoidReduction;
  INHERITS DuraReduction;

  REDEFINE METHOD CREATE (BaseClass,
                          DerivedClass: DuraClassDefinition);
END PolymorphismVoidReduction;
```

It is of course a very simple form of reduction, since it reuses the default code generation schemes (See method *DuraReductions.DuraReduction.GenerateReductionCode* on page 823) without alteration.

834 ⟨PolymorphismVoidReduction *class implementation* 834⟩≡

```
CLASS PolymorphismVoidReduction;
  INHERITS DuraReduction;

  VAR
    TheBaseClass, TheDerivedClass: DuraClassDefinition;

  REDEFINE METHOD CREATE (BaseClass, DerivedClass: DuraClassDefinition);
    BEGIN
    BASE;
    TheBaseClass ← BaseClass;
    TheDerivedClass ← DerivedClass;
    END CREATE;

  REDEFINE METHOD Dump(Output: OutputStream);
    BEGIN
    Output.WriteString (``Polymorphic red: ” +
             TheDerivedClass.DotImage + ``->” +
                        TheBaseClass.DotImage);
    END Dump;

  REDEFINE METHOD IsEqual (Other: DuraReduction): BOOLEAN;
    BEGIN
    IF Other ≠ Λ THEN
      WHAT Other OF
        IN PolymorphismVoidReduction:
          RESULT ← TheBaseClass = TAG.TheBaseClass;
          END;
       ELSE
        END;
      END;
    END IsEqual;

  ⟨Implicit, as redefined by PolymorphismVoidReduction 835a⟩

END PolymorphismVoidReduction;
```

A `PolymorphismVoidReduction` is implicit – understand: can be simplified – if by removing this reduction, we do not change the parser's behaviour in terms of the cut and recovery levels.

835a ⟨`Implicit`, *as redefined by* `PolymorphismVoidReduction` 835a⟩≡
```
REDEFINE METHOD Implicit: BOOLEAN;
  BEGIN
  RESULT ← (TheBaseClass.CutLevel =
                        TheDerivedClass.CutLevel) ∧
           (TheBaseClass.RecoveryLevel =
                         TheDerivedClass.RecoveryLevel);
  END Implicit;
```

7.26.5 `IndicatorReduction`

835b ⟨`IndicatorReduction` *class definition* 835b⟩≡
```
CLASS IndicatorReduction;
  INHERITS DuraReduction;

  REDEFINE METHOD CREATE (Attr: InterAttribute;
                          Value: INTEGER);
END IndicatorReduction;
```

`IndicatorReduction` is used in the case of a grammatical `CASE` expression (See page 261). For instance, given the expression:

```
GRAMMAR
  CASE TheForm OF
    70:
      Move Expression To Designator
      END;
    71:
      Expression Arrow Designator
      END;
    72:
      Designator Becomes Expression
      END;
    END;
```

`DURA` will consider a flat representation of the form:

$$
\begin{aligned}
\mathit{Synth} \; ::= \; & \mathit{Move\ Expression\ To\ Designator} \mid \\
& \mathit{Expression\ Arrow\ Designator} \mid \\
& \mathit{Designator\ Becomes\ Expression}
\end{aligned}
$$

and it will attach an instance of `IndicatorReduction` to each of *Synth*'s right hand sides, indicating that `TheForm` must be given the value resp. 70, 71 and 72.

836 ⟨`IndicatorReduction` *class implementation* 836⟩≡

```
CLASS IndicatorReduction;
  INHERITS DuraReduction;

  VAR
    TheAttr: InterAttribute;
    TheValue: INTEGER;

  REDEFINE METHOD CREATE (Attr: InterAttribute;
                          Value: INTEGER);
    BEGIN
    BASE;
    TheAttr ← Attr;
    TheValue ← Value;
    END CREATE;

  REDEFINE METHOD Dump (Output: OutputStream);
    BEGIN
    Output.WriteString (‘‘ Indicator reduction: ” +
                        TheAttr.FullName + ‘‘ Value:”);
    Output.WriteInt (TheValue, 0);
    Output.WriteLn;
    END Dump;
```

⟨*Generating code for an* `IndicatorReduction` 837a⟩
⟨*Checking wether two* `IndicatorReduction`*'s are equal* 837b⟩

END `IndicatorReduction;`

Generating code for an indicator is merely a question of using the corresponding meta-attribute (See method *DuraIntermediateFormat.InterAttribute.GenerateMetaAttribute* on page 738), and to provide it with the value `TheValue`.

837a ⟨*Generating code for an* `IndicatorReduction` 837a⟩≡

```
REDEFINE METHOD GenerateFetchCode (Output: DuraOutputStream;
                                   RHS: FlatRHS);
  BEGIN
  TheAttr.GenerateMetaAttribute (Output);
  Output.WriteString (“.Set(”);
  Output.WriteInt (TheValue, 0);
  Output.WriteString (“); --”);
  Dump (Output);
  END GenerateFetchCode;
```

837b ⟨*Checking wether two* `IndicatorReduction`*'s are equal* 837b⟩≡

```
REDEFINE METHOD IsEqual (Other: DuraReduction): BOOLEAN;
  BEGIN
  IF Other ≠ Λ THEN
    WHAT Other OF
      IN IndicatorReduction:
        RESULT ← (TheAttr = TAG.TheAttr) ∧
                 (TheValue = TAG.TheValue);
        END;
     ELSE
      END;
    END;
  END IsEqual;
```

7.26.6 `FinalReduction`

`FinalReduction` is attached to the RHS of the main grammar rule, which reduces the main non-terminal – the target of our grammar – just before the parser ends.

837c ⟨`FinalReduction` *class definition* 837c⟩≡

```
CLASS FinalReduction;
  INHERITS DuraReduction;
END FinalReduction;
```

838a ⟨FinalReduction *class implementation* 838a⟩≡

```
CLASS FinalReduction;
  INHERITS DuraReduction;

  REDEFINE METHOD Dump (Output: OutputStream);
    BEGIN
    Output.WriteString ("‘ Final reduction”);
    END Dump;

  ⟨Code generation, as redefined by FinalReduction 838b⟩

  REDEFINE METHOD IsEqual (Other: DuraReduction): BOOLEAN;
    BEGIN
    IF Other ≠ Λ THEN
      WHAT Other OF
        IN FinalReduction:
          RESULT ← TRUE;
          END;
       ELSE
        END;
      END;
    END IsEqual;

END FinalReduction;
```

GenerateReductionCode overrides its inherited version (See page 823) and generates basically nothing ! It leaves the root non-terminal of the grammar on the LR stack, and the parser understands its structure and extracts the target non-terminal from it (See page 897).

838b ⟨*Code generation, as redefined by* FinalReduction 838b⟩≡

```
REDEFINE METHOD GenerateReductionCode (Output: DuraOutputStream;
                                       RHS: FlatRHS);
  BEGIN
  Output.WriteLine ("‘  BEGIN”);
  END GenerateReductionCode;
```

7.27 DuraBitMap

The `DuraBitMap` module implements bitmaps, with an interface which is quite similar to the `BitMap` module provided by YAFL's standard library. Its implementation is optimized for sparse and large bitmaps and the management of large numbers of very similar bitmaps.

I marked this chunk as important because it illustrates an idea I believe is important: implementation still matters. There is nothing thrilling about managing bitmaps. There are several rather well-known ways of representing bitmaps, but none of the ones I've been able to consider do address our need acceptably. The added value of this chapter lies in a working implementation of mundane facilities in a stressed environment, where we must manage amounts of data such that memory and performance become a major concern.

Please read forward to discover the mystery of the implementation of these magnificent bitmaps. You will have to go beyond the definition to guess how this happened.

⟨*Definition* ⟩≡
```
DEFINITION MODULE DuraBitMap;

IMPORT HashElement;
IMPORT HashSpace;
IMPORT BitMap;

  ⟨DuraBitMap class definition 843⟩
  ⟨BitMapPage class definition 852a⟩
  ⟨Page dictionary definition 855c⟩

END DuraBitMap;
```

7.27.1 The basic idea

Well, since you got as far as this, you probably deserved to get the whole story.

Most if not all the algorithms that deal with language recognizers, at the syntactical or lexical level, are based on *sets*. Sets of tokens, sets of items, sets of states, etc... As a matter of fact, it is a slight abuse of language to claim we are dealing with sets. We are more precisely dealing with sub-sets, out of a common reference set. When we consider a set of bananas, we are in fact dealing with a subset of the set of all bananas.

A naive implementation of such sub-sets can be based on a bitmap, where every element of the reference set is represented by a single bit. In such an implementation, the size of the data structure depends on the size of the reference set, with no regard for the size of the actual subsets. On the other hand, all common set operation can be performed easily and efficiently. For instance, checking whether an element of the reference set is or is not in a given sub-set, it is sufficient to check the corresponding bit in the bitmap.

When dealing with large parsers and large lexers, the reference sets happen to be quite large (over 15000 elements). Furthermore, the parser generator might have to handle over a few thousands of such large sub-sets. A naive implementation based on plain bitmaps (one bit per element, no compression) as in [45] would be inefficient and memory-hungry.

The implementation described in this module is based on two basic assumptions that are applicable to the usage of bitmaps within DURA's context, but that might not be applicable in a different environment:

- The bitmaps will usually be sparse. For instance, when dealing with the transformation of a NFA[12] made of over 10000 states into a semantically equivalent DFA[13], we will seldom need sets that contain more than a dozen, perhaps a hundred elements. In other words, our bitmaps will be almost entirely filled with zero's (See table 7.2).

- Many bitmaps will be similar, in the sense that some bitmaps will only differ from each other by a few bits. Stated differently, this means that when dealing with state or item sets in a language processor, the distribution of actually used sub-sets with regard to the entire powerset will be very specific, and we will be able to divide our sets in classes based on Hamming's distance[14].

[12] Non-deterministic finite automaton

[13] Needless to say, deterministic finite automaton

[14] Hamming's distance between two sets (in this specific case, bitmaps) is the number of elements (bits) by which they differ. It has all the required properties to be a distance, namely, $Hamming(A, A) = 0 \,\forall\, A$, $Hamming(A, B) \geq 0 \,\forall\, A, B$ and $Hamming(A, C) \leq Hamming(A, B) + Hamming(B, C) \,\forall\, A, B, C$.

Table 7.2: Relative sizes of DFA and NFA's for several lexical analyzers, with distribution parameters for the size of the bitmaps

	NFA states	DFA states	Min	Max	Average	StdDev
Cobol	11105	1809	1	1890	31.40	45.66
PL/1	8352	1450	1	1613	26.43	45.47
Fortran	3826	738	1	722	49.94	36.70
ObjectZ	2408	1025	1	669	9.78	22.69
C++	793	405	1	286	7.74	14.76

One might be tempted to use a more flexible and space-efficient data structure than plain bitmaps such as sorted lists, but that might have induced a space penalty if some of the sets happen not to be as sparse as originally thought[15], and a significant performance penalty when dealing with very common and heavily used primitives such as union ($A \cup B$), ownership ($e \in A$) and inclusion testing ($A \subset B$).

So, instead of using plain bitmaps, we divide these bitmaps into fixed size pages of 1024 bits, implemented by the `BitMapPage` class (See page 852). These pages are only allocated if at least one of their bits must be set. By doing so, a bitmap from which 5 bits are set within a total of 15000 will allocate at most 5 pages (if each of the bits is on a different page) instead of 15.

When dealing with sets of LR items, given the way they are numbered, we can even expect the probability of different bits to be on the same page to be rather high in some cases. For instance, consider the following grammar extract:

$$\begin{array}{lll} M & ::= & A_1\ A_2\ \ldots A_n \ \mid \\ & & B_1\ B_2\ \ldots B_m \ \mid \\ & & C_1\ C_2\ \ldots C_p \end{array}$$

The LR items are numbered in a rather straightforward way:

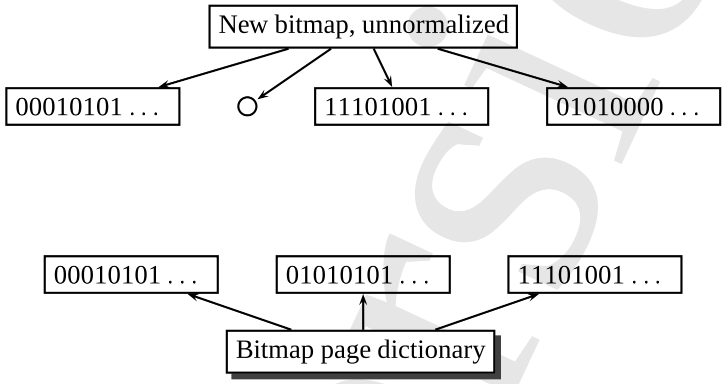
$$\begin{array}{lll} M & ::= & \underline{\textbf{(k)}}\ A_1\ \underline{\textbf{(k+1)}}\ A_2\ \underline{\textbf{(k+2)}} \ldots A_n\ \underline{\textbf{(k+n)}} \ \mid \\ & & \underline{\textbf{(k+n+1)}}\ B_1\ \underline{\textbf{(k+n+2)}}\ B_2\ \underline{\textbf{(k+n+3)}} \ldots B_m\ \underline{\textbf{(k+n+m+1)}} \ \mid \\ & & \underline{\textbf{(k+n+m+2)}}\ C_1\ \underline{\textbf{(k+n+m+3)}}\ C_2\ \underline{\textbf{(k+n+m+4)}} \ldots \underline{\textbf{(k+n+m+p+2)}} \end{array}$$

Any LR state which includes item (k) includes items $(k+n+1)$ and $(k+n+m+2)$ as well. Given the relative orders of magnitude of the page size (1024) and n, m and p above, we might expect that the items that refer to the beginning of a given grammar rule have a high probability to be stored on the same page, when the item sets will be turned into LR states.

I also implemented a normalization process, that stores all unique pages in a dictionary, implemented by the `PageDictionary` class (See page 855). When a page is being normalized, it refers to pages that are stored in the dictionary, and which can be referred to by other normalized pages.

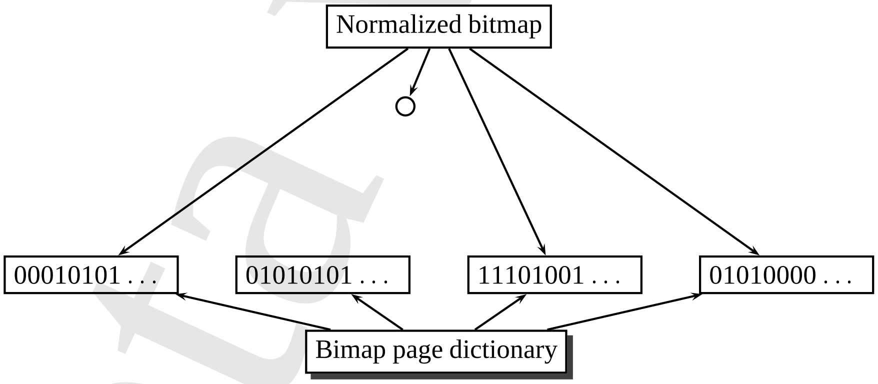
[15] One could think of storing such not-so-sparse bitmaps as their complements, but we did not investigate this option any further, considering the linked list approach as doomed to offer poor overall performance.

Figure 7.17 Unnormalized bitmap



For instance, consider figure 7.17. The unnormalized bitmap owns pages that are already present in the dictionary before the normalization.

Figure 7.18 Normalized bitmap



After normalization, the page refers to the dictionary's shared pages, and those pages that were not in the dictionary yet are added.

This whole dictionary business has three major implications:

- A normalized page must be unmutable, since any alteration might have side effects on other bitmaps that refer to that same page. When a bitmap needs to update one of its normalized pages, it must first reallocate an unormalized copy of the original page.
- By sharing pages, one can expect the total memory consumption to be much lower, specially if one expects numerous bitmaps to be similar, that is, to have several pages in common.
- Comparing two normalized bitmaps for equality is as simple as verifying whether they refer to the same pages in the same order. The dictionary and the normalization process provide an equivalence between a reference and a value semantics for bitmap pages. Hence, checking whether the pages referred to by two references are the same (have the same bits set) is equivalent to checking whether the two references are equal.

The results of this construction are described in table 7.3.

```
⟨Implementation ⟩≡
  IMPLEMENTATION MODULE DuraBitMap;

  IMPORT BitMap;
  FROM Streams IMPORT StdOut;
  IMPORT SYSTEM;

    ⟨DuraBitMap class implementation 844a⟩
    ⟨BitMapPage class implementation 852b⟩
    ⟨Page dictionary implementation 856a⟩

  END DuraBitMap;
```

7.27.2 Bitmaps

843
```
⟨DuraBitMap class definition 843⟩≡
  CLASS DuraBitMap;
    INHERITS HashElement;

    ⟨DuraBitMap CREATE method definition 844d⟩

    ⟨Element manipulation method definitions 846d⟩
    ⟨Set operations method definitions 846b⟩
    ⟨UsedPages method definition 851d⟩

    METHOD Normalize;

  END DuraBitMap;
```

844a ⟨DuraBitMap *class implementation* 844a⟩≡

```
CLASS DuraBitMap;
  INHERITS HashElement;

  CONST
    PageSize = BitMapPage.PageSize;

  VAR
    ⟨DuraBitMap attributes 844b⟩

  ⟨DuraBitMap CREATE method implementation 844e⟩
  ⟨Element manipulation method implementations 847a⟩
  ⟨Set operations method implementations 846c⟩

  ⟨Normalization methods 845a⟩
  ⟨Hash function method 846a⟩
  ⟨UsedPages method implementation 851e⟩

END DuraBitMap;
```

`TheSize` holds the number of bits that can be stored in the bitmap, that is, the size of the reference set of which the bitmap is a subset; not to be confused with the number of bits that are set at a given time. The bitmap's size is a constant for its entire lifetime, while bits can be set or cleared dynamically, and the number of bits that are set can vary.

844b ⟨DuraBitMap *attributes* 844b⟩≡

```
TheSize: INTEGER;
```

`ThePage` is the array of pages. However, the pages themselves will only be allocated on demand. Hence, the array of pages is allocated, but all its entries are initialized with `VOID`, indicating that no bit is set.

844c ⟨DuraBitMap *attributes* 844b⟩+≡

```
ThePage: ARRAY OF BitMapPage;
```

When creating a bitmap, the number of bits must be provided. The number of pages is computed accordingly.

844d ⟨DuraBitMap CREATE *method definition* 844d⟩≡

```
REDEFINE METHOD CREATE(Size: INTEGER);
```

`ThePage` is `CREATE`d with a size of $\lceil \frac{Size}{PageSize} \rceil$.

844e ⟨DuraBitMap CREATE *method implementation* 844e⟩≡

```
REDEFINE METHOD CREATE(Size: INTEGER);
  BEGIN
  ASSERT Size > 0; -- Just in case...
  BASE;
  TheSize ← Size;
  ThePage.CREATE ((Size + PageSize - 1) / PageSize);
  END CREATE;
```

`NormalizedPage` indicates whether a given page is normalized or not, based on the assumption that an empty page, represented by a `VOID` reference, is normalized.

845a ⟨*Normalization methods* 845a⟩≡

```
METHOD NormalizedPage (PageNr: INTEGER): BOOLEAN;
  BEGIN
  RESULT ← (ThePage_PageNr = Λ) ∨
              (ThePage_PageNr.IsNormalized);
  END NormalizedPage;
```

Normalizing a `DuraBitMap` instance simply means that instead of using our own pages, we store our pages in the page dictionary, where they can be shared among several normalized bitmaps.

845b ⟨*Normalization methods* 845a⟩+≡

```
METHOD Normalize;
BEGIN
FOR i ∈ 0 .. ThePage.SIZE - 1 | ¬ NormalizedPage(i) DO
  ThePage_i ← ThePage_i.Normalize;
  ⟨Make sure that ThePage[i] is indeed normalized 845c⟩
  END;
END Normalize;
```

Normalization is an idempotent operation, that is, when a page has been stored in the page dictionary, looking for this page in the dictionary should return that very page.

It is also tempting to state that `ThePage[i]` cannot be `VOID` after being normalized. However, one can turn a bit on, causing the page to be allocated, turn it off, leaving the page allocated, but no bit on anymore. Such a page is not `VOID`, not normalized, but normalizing it yields `VOID`.

845c ⟨*Make sure that* `ThePage[i]` *is indeed normalized* 845c⟩≡

```
ASSERT ThePage_i ≠ Λ ⇒
     ThePage_i = ThePage_i.Normalize;
```

Computing a `DuraBitMap` hash value is based on its pages hash values, which can be computed quite efficiently, since they are cached and don't have to be reevaluated on each invocation (See method *DuraBitMap.BitMapPage.HashValue* on page 853).

The parameters of the hashing function are based on my lack of expertise in the field, and the vague reminiscence of the usefulness of mutually prime numbers in this context, hence 3 and 17. Someday, I'll read Knuth's reference book [88] again, and consider improving this hashing function if required. The only copy I've ever had of this excellent book is now lost in a dark forest in the Belgian Ardennes (Hard to believe, but dramatically true).

846a ⟨*Hash function method* 846a⟩≡
```
REDEFINE METHOD HashValue: INTEGER;
  BEGIN
  RESULT ← 1019;
  FOR Page ∈ ThePage DO
    IF Page = Λ THEN
      RESULT ← RESULT + RESULT + 17;
     ELSE
      RESULT ← SYSTEM.BitXor (RESULT * 3, Page.HashValue);
      END;
    END;
  END HashValue;
```

Well, I guess there is not much to say about this method.

846b ⟨*Set operations method definitions* 846b⟩≡
```
METHOD Size: INTEGER;
  POST
    NO_CHANGE;
```

846c ⟨*Set operations method implementations* 846c⟩≡
```
METHOD Size: INTEGER;
  BEGIN
  RESULT ← TheSize;
  END Size;
```

The `Get` method can be used to test whether a given bit in a `DuraBitMap` instance is set or not.

846d ⟨*Element manipulation method definitions* 846d⟩≡
```
METHOD Get(Idx: INTEGER): BOOLEAN;
  POST
    NO_CHANGE;
```

It simply finds the proper page, checks whether it is allocated or not, and checks the proper bit in that page.

847a ⟨*Element manipulation method implementations* 847a⟩≡

```
METHOD Get(Idx: INTEGER): BOOLEAN;
  VAR
    p: BitMapPage;
  BEGIN
  ASSERT Idx ≥ 0;
  ASSERT Idx < TheSize;
  p ← ThePage[Idx / PageSize];
  RESULT ← (p ≠ Λ) ∧
                p.Get(Idx MOD PageSize);
  END Get;
```

The `Set` method sets a bit in a `DuraBitMap` instance.

847b ⟨*Element manipulation method definitions* 846d⟩+≡

```
METHOD Set(Idx: INTEGER);
```

`Set` first checks whether the bit has already been set, in which case it does nothing.

847c ⟨*Element manipulation method implementations* 847a⟩+≡

```
METHOD Set(Idx: INTEGER);
  VAR
    PageNr: INTEGER;
  BEGIN
  IF ¬ Get(Idx) THEN
    PageNr ← Idx / PageSize;
    ⟨Ensure that page PageNr is kosher 847d⟩
    ThePage_PageNr.Set(Idx MOD PageSize);
    END;
  END Set;
```

The page on which a bit must be set must be allocated.

847d ⟨*Ensure that page* `PageNr` *is kosher* 847d⟩≡

```
IF ThePage_PageNr = Λ THEN
  ThePage_PageNr.CREATE;
```

If the page is normalized, it is not mutable anymore, since it can be shared among several normalized bitmaps. Hence, it must be cloned first. The cloned copy is not normalized itself. It can be normalized again by calling `Normalize`, which only normalizes unnormalized pages within a bitmap.

This chunk shows a possible flaw in the normalization scheme. If a bitmap is normalized, and happens to be the only owner of a given page (understand: the only bitmap that refers to that page) attempting to perform a change to that page will cause an unneeded reallocation. As we'll see in the closing paragraph of this section, we'll structure our code in such a way that a bitmap life cycle prevents such reallocation to occur in practice.

This chunk calls `Clone` (See page 855) rather than the predefined `CLONE` method, but this is essentially an historical remaining of some complex debugging session.

848a ⟨*Ensure that page* `PageNr` *is kosher* 847d⟩+≡
```
  ELSIF ThePage_PageNr.IsNormalized THEN
    ThePage_PageNr ← ThePage_PageNr.Clone;
    ASSERT ¬ ThePage_PageNr.IsNormalized;
    END;
```

`Clear` clears a bit of a `DuraBitMap` instance.

848b ⟨*Element manipulation method definitions* 846d⟩+≡
```
METHOD Clear(Idx: INTEGER);
```

Similarly to what has been described for the `Set` method above, if clearing a bit implies modifying a normalized page, it must be cloned first so that a non-normalized copy is used. In any case, if first checks whether the bit to clear is not already cleared but calling `Get` (See page 846), in which case it simply does nothing.

848c ⟨*Element manipulation method implementations* 847a⟩+≡
```
METHOD Clear(Idx: INTEGER);
  VAR
    PageNr: INTEGER;
  BEGIN
  IF Get(Idx) THEN
    PageNr ← Idx / PageSize;
    ASSERT ThePage_PageNr ≠ Λ; -- Implied by Get(Idx)
    IF ThePage_PageNr.IsNormalized THEN
      ThePage_PageNr ← ThePage_PageNr.Clone;
      ASSERT ¬ ThePage_PageNr.IsNormalized;
      END;
    ThePage_PageNr.Clear(Idx MOD PageSize);
    END;
  END Clear;
```

`IsEqual` checks whether two `DuraBitMap` instances are equivalent, that is:

- They have exactly the same size
- They have exactly the same bits set.

848d ⟨*Set operations method definitions* 846b⟩+≡
```
METHOD IsEqual (Other: DuraBitMap): BOOLEAN;
```

Equality is implemented by first normalizing both bitmaps, that is, by ensuring that all the pages of the two bitmaps are normalized.

Then, we use the equivalence between the reference and the value semantics for normalized pages to check for the equality page per page.

849a ⟨*Set operations method implementations* 846c⟩+≡
```
METHOD IsEqual (Other: DuraBitMap): BOOLEAN;
  BEGIN
  IF Other ≠ Λ THEN
    IF Other.Size = TheSize THEN
      Normalize;
      Other.Normalize;
      ASSERT ThePage.SIZE = Other.ThePage.SIZE;
      RESULT ← ∀ i ∈ 0 .. ThePage.SIZE - 1 •
                  ThePage_i = Other.ThePage_i;
      END;
    END;
  END IsEqual;
```

`Intersects` indicates whether two `DuraBitMap` instances have a non-empty intersection.

849b ⟨*Set operations method definitions* 846b⟩+≡
```
METHOD Intersects (Other: DuraBitMap): BOOLEAN;
```

Checking for a non void intersection translates into checking for that same intersection page per page.

849c ⟨*Set operations method implementations* 846c⟩+≡
```
METHOD Intersects (Other: DuraBitMap): BOOLEAN;
  BEGIN
  IF (Other ≠ Λ) ∧ (Other.ThePage ≠ Λ) THEN
    ASSERT Size = Other.Size;
    ASSERT ThePage.SIZE = Other.ThePage.SIZE;
    FOR i ∈ 0 .. ThePage.SIZE - 1 WHILE ¬ RESULT DO
      ⟨Check for page number i 849d⟩
      END;
    END;
  END Intersects;
```

When checking whether two pages intersect or not, we simply discard the case where one of the two pages is void, since $A \cap \phi = \phi$ no matter what A stands for, and $\phi \cap \phi = \phi$.

849d ⟨*Check for page number* `i` 849d⟩≡
```
IF (ThePage_i ≠ Λ) ∧ (Other.ThePage_i ≠ Λ) THEN
  RESULT ← ThePage_i.Intersects (Other.ThePage_i);
  END;
```

The `Or` method is used to compute the logical disjunction between two `DuraBitMap` instances. When executing `One.Or(Two)`, what actually gets executed is $One \leftarrow One \cup Two$.

850a ⟨*Set operations method definitions* 846b⟩+≡
```
METHOD Or (Other: DuraBitMap);
```

850b ⟨*Set operations method implementations* 846c⟩+≡
```
METHOD Or (Other: DuraBitMap);
  VAR
    OtherPage : BitMapPage;
  BEGIN
  IF (Other ≠ Λ) ∧ (Other.ThePage ≠ Λ) THEN
    ASSERT Size = Other.Size;
    ASSERT ThePage.SIZE = Other.ThePage.SIZE;
    FOR i ∈ 0 .. ThePage.SIZE - 1 DO
      OtherPage ← Other.ThePage_i;
      ⟨Compute the disjunction between OtherPage and ThePage[i] 850c⟩
      END;
    END;
  END Or;
```

First, we consider the case where at least one of the two pages is void. It is merely a question of sharing a reference to a notoriously unmutable (understand: normalized) page, or cloning a non-normalized page.

850c ⟨*Compute the disjunction between* `OtherPage` *and* `ThePage[i]` 850c⟩≡
```
IF ThePage_i = Λ THEN
  IF OtherPage ≠ Λ THEN
    IF OtherPage.IsNormalized THEN
      ThePage_i ← OtherPage;
     ELSE
      ThePage_i ← OtherPage.Clone;
      END;
    END;
```

Then, we'll consider the case where the two pages are non-void. In order to avoid possibly unnecessary reallocations (see below), we first ensure that `Or`ing the two pages would result in changing something: $A \subset B \Leftrightarrow B \cup A = B$.

850d ⟨*Compute the disjunction between* `OtherPage` *and* `ThePage[i]` 850c⟩+≡
```
 ELSIF OtherPage ≠ Λ THEN
  ASSERT ThePage_i ≠ Λ;
  IF ¬ ThePage_i.Includes (OtherPage) THEN
```

We must check whether `ThePage[i]` is normalized, in which case it must be reallocated. Then, we can rely on the existing `Or` method provided by the `BitMap` class extracted from YAFL's standard library.

851a ⟨*Compute the disjunction between* `OtherPage` *and* `ThePage[i]` 850c⟩+≡
```
      IF ThePage_i.IsNormalized THEN
        ThePage_i ← ThePage_i.Clone;
        END;
      ASSERT ¬ ThePage_i.IsNormalized;
      ThePage_i.Or (OtherPage);
      END;
    END;
```

`UsedBits` returns the number of bits that are set within a `DuraBitMap` instance.

851b ⟨*Set operations method definitions* 846b⟩+≡
```
  METHOD UsedBits: INTEGER;
    POST
      RESULT ≥ 0,
      RESULT ≤ Size;
```

The implementation is quite straightforward...

851c ⟨*Set operations method implementations* 846c⟩+≡
```
  METHOD UsedBits: INTEGER;
  BEGIN
  FOR Page ∈ ThePage | Page ≠ Λ DO
    RESULT ← RESULT + Page.UsedBits;
    END;
  END UsedBits;
```

851d ⟨`UsedPages` *method definition* 851d⟩≡
```
  METHOD UsedPages: INTEGER;
```

851e ⟨`UsedPages` *method implementation* 851e⟩≡
```
  METHOD UsedPages: INTEGER;
    BEGIN
    FOR Page ∈ ThePage | Page ≠ Λ DO
      RESULT ← RESULT + 1;
      END;
    END UsedPages;
```

7.27.3 Bitmap pages

Well, this was the way things happened at the `DuraBitMap` level. Let's now investigate how individual pages are managed.

852a ⟨`BitMapPage` *class definition* 852a⟩≡

```
CLASS BitMapPage;
  INHERITS BitMap;

  REDEFINE METHOD CREATE;
END BitMapPage;
```

852b ⟨`BitMapPage` *class implementation* 852b⟩≡

```
CLASS BitMapPage;
  INHERITS BitMap;

  ⟨PageSize constant declaration 852c⟩
  ⟨Creating a BitMapPage 852d⟩
  ⟨Hash value management 853a⟩
  ⟨Normalization features 853b⟩
  ⟨Cloning a page 855a⟩

END BitMapPage;
```

Evaluating the ideal size for pages is a matter of making the right compromise, between smaller pages in order to increase the reusability rate among bitmaps, and larger pages to reduce the total number of pages to allocate, hence, reducing the impact of allocation costs.

1024 sounds like an acceptable value: a bitmap of 1024 elements requires 128 bytes while YAFL's current implementation requires 16 additional bytes for every allocation. Hence, the allocation overhead when compared to monolithic pages is 12.5%.

It is possible that another page size might provide even better results, but the average gain depicted in table 7.3 is good enough for any practical purpose, and I did not investigate this issue any further.

852c ⟨`PageSize` *constant declaration* 852c⟩≡

```
CONST
  PageSize = 1024;
```

852d ⟨*Creating a* `BitMapPage` 852d⟩≡

```
REDEFINE METHOD CREATE;
  BEGIN
  TheHashValue ← -1;
  BASE (PageSize);
  END CREATE;
```

When the page is normalized, its hash key value is cached in an attribute, since `THIS` cannot change anymore. As long as the page is not normalized, the inherited `HashValue` method is called instead.

853a ⟨*Hash value management* 853a⟩≡

```
VAR
  TheHashValue: INTEGER;

REDEFINE METHOD HashValue: INTEGER;
BEGIN
IF TheIsNormalized THEN
  IF TheHashValue < 0 THEN
    TheHashValue ← SYSTEM.ABS(BASE);
    END;
  RESULT ← TheHashValue;
 ELSE
  RESULT ← SYSTEM.ABS(BASE);
  END;
END HashValue;
```

`TheIsNormalized` indicates whether THIS is normalized:

- THIS is referenced by `PageDictionary`
- THIS is unmutable. One cannot set or clear a bit in a normalized page, since several references to it might exist, and they might rely on its current value. Any reference that attempts to mute a normalized `BitMapPage` should first reallocate it (and actually does, please see relevant chunks above)

853b ⟨*Normalization features* 853b⟩≡

```
VAR
  TheIsNormalized: BOOLEAN;

METHOD IsNormalized: BOOLEAN;
  BEGIN
  RESULT ← TheIsNormalized;
  END IsNormalized;

⟨Normalize method implementation 853c⟩
⟨Make sure that a normalized page is unmutable 854c⟩
```

When normalizing a page, one first checks whether it is already normalized. Normalization is an idempotent operation.

853c ⟨`Normalize` *method implementation* 853c⟩≡

```
METHOD Normalize: BitMapPage;
  BEGIN
  IF TheIsNormalized THEN
    RESULT ← THIS;
```

The normalized representation of a page where no bit is set is VOID.

854a ⟨Normalize *method implementation* 853c⟩+≡
```
  ELSIF UsedBits > 0 THEN
    ⟨THIS is not empty 854b⟩
    END;
ASSERT RESULT ≠ Λ ⇒ RESULT.TheIsNormalized;
END Normalize;
```

When normalizing a non-empty page, one first have to check whether an equivalent page is referred by the `PageDictionary` once class (See the next page). If it is not the case, THIS is added to `PageDictionary`.

854b ⟨THIS *is not empty* 854b⟩≡
```
RESULT ← PageDictionary.FindPage (THIS);
IF (RESULT = Λ) THEN
  RESULT ← THIS;
  TheIsNormalized ← TRUE;
  TheHashValue ← -1;
  PageDictionary.StorePage(THIS);
  END;
```

The three following methods are redefined in order to ensure that a normalized `BitMapPage` cannot be changed anymore. In practice, none of these assertions has ever caused a system to stop so far, but paranoia is a safe move...

854c ⟨*Make sure that a normalized page is unmutable* 854c⟩≡
```
REDEFINE METHOD Set(Index: INTEGER);
  BEGIN
  ASSERT ¬ TheIsNormalized;
  BASE (Index);
  END Set;

REDEFINE METHOD Clear(Index: INTEGER);
  BEGIN
  ASSERT ¬ TheIsNormalized;
  BASE (Index);
  END Clear;

REDEFINE METHOD Or (Other: BitMap);
  BEGIN
  ASSERT ¬ TheIsNormalized;
  BASE (Other);
  END Or;
```

This is bizarre: when one clones a page, the newly allocated page cannot be normalized, since it would contradict the basic property of normalized pages which is uniqueness.

855a ⟨*Cloning a page* 855a⟩≡
```
REDEFINE METHOD CLONE (Other: BitMapPage);
  BEGIN
  BASE(Other);
  TheIsNormalized ← FALSE;
  TheHashValue ← -1;
  END CLONE;
```

The `Clone` method has been defined in a moment of ultimate doubts about how the `CLONE` predefined method worked . . .

855b ⟨*Cloning a page* 855a⟩+≡
```
METHOD Clone: BitMapPage;
  BEGIN
  RESULT ← THIS.CLONE;
  ASSERT RESULT ≠ THIS;
  ASSERT ¬ RESULT.TheIsNormalized;
  END Clone;
```

7.27.4 The page dictionary

After dealing with pages, we will now describe how the system keeps track of all distinct pages in a dictionary. At this level, it is amazingly simple: the `PageDictionary` once class merely supports two primitives: searching for a given page, and storing a new page within the dictionary. Hashing is used for fast retrieval of an existing page in our `PageDictionary`. Alternatively, we could have used a tree structure, as in module `DuraIntDictionary` (See page 763), but I must confess that hashing is a personal favorite.

The dictionary inherits from `HashSpace`, and is created with an initial size of 4096. It resizes itself automatically when half full, in order to reduce the likeliness of collision as far as possible.

855c ⟨*Page dictionary definition* 855c⟩≡
```
ONCE CLASS PageDictionary;
  INHERITS HashSpace (BitMapPage);

  REDEFINE METHOD CREATE;
END PageDictionary;
```

856a ⟨*Page dictionary implementation* 856a⟩≡

```
ONCE CLASS PageDictionary;
  INHERITS HashSpace (BitMapPage);

  REDEFINE METHOD CREATE;
    BEGIN
    BASE(4096);
    END CREATE;

  ⟨Finding a page 856b⟩
  ⟨Storing a new page 856c⟩

END PageDictionary;
```

When looking for a page within the dictionary, one must successively check for all the entries that collide with the hash value attached to the page we're looking for until we find a matching page.

YAFL

To check whether two `BitMapPage` instances are the same, we use the `IsEqual` method inherited from the `BitMap` class provided by the standard library, and which is highly optimized to check for equality without extracting all the bits one by one.

856b ⟨*Finding a page* 856b⟩≡

```
METHOD FindPage (p: BitMapPage): BitMapPage;
  BEGIN
  RESULT ← Find(p.HashValue);
  WHILE (RESULT ≠ Λ) ∧ (¬ RESULT.IsEqual(p)) DO
    RESULT ← Next;
    END;
  END FindPage;
```

Storing a page is simply a matter of checking for the dictionary's size and updating it if required, and call the `Store` method provided by the `HashTable` class provided by YAFL's standard library.

856c ⟨*Storing a new page* 856c⟩≡

```
METHOD StorePage (p: BitMapPage);
  VAR
    Flag: BOOLEAN;
  BEGIN
  ASSERT p.IsNormalized;
  ⟨Make sure that the page p is not in the dictionary yet 857b⟩
  ⟨Check for the dictionary size 857a⟩
  Flag ← Store (p);
  ⟨Ensure everything worked as predicted 857c⟩
  END StorePage;
```

In order to improve the performance, given our rather naive and inefficient collision handling, we ensure that the hash table remains always at least twice as big as the number of elements it holds.

857a ⟨*Check for the dictionary size* 857a⟩≡
```
IF 2 * Stored > AllocatedSize THEN
  Resize (AllocatedSize * 2);
  END;
```

857b ⟨*Make sure that the page* `p` *is not in the dictionary yet* 857b⟩≡
```
ASSERT FindPage(p) = Λ;
```

857c ⟨*Ensure everything worked as predicted* 857c⟩≡
```
ASSERT Flag;
ASSERT p = FindPage(p);
```

7.27.5 Results and conclusions

The careful reader might have noticed that the normalization process as described here works best if the bitmaps follow a well-defined pattern, namely:

- They are computed entirely without being normalized. At this stage, they can be altered in any way without extra cost.
- Then, as soon as they have been normalized once (for instance, when they are compared for equality with other bitmaps), they are not altered anymore. Any exception to this rule would induce potentially unneeded reallocations.

In practice, the way these bitmaps have been used in `DURA` and `lexyt` matches this pattern very closely. It is true however that naive usage of these classes without paying attention to this very specific property might result in pathetic results, due to systematic reallocation of previously normalized pages.

After producing what I thought was a great piece of software, I started considering ways of demonstrating its usefulness and valuable properties. I first tried to plot distribution of Hamming's distance between bitmaps, but that gave very little true information. I also tried to compare some of the results with a theoretically predictable distribution, to emphasize how these constructs were particularly suited to the task at hand. I then realize that this so-called "theoretically predictable" distribution was by far more complex to compute than originally thought.

Well, for all these reasons, I simply preferred to measure the impact of the division of bitmaps into pages and the normalization process in terms of memory occupation. After all, the rest is little more than theoretical *pilpul*[16].

The results are summarized in the table displayed in table 7.3.

[16] *pilpul* is a talmudic term that denotes formal arguments with little if any true meaning or usefulness.

Table 7.3: Statistics about the space optimization of DuraBitMap

Measured system	COBOL lexer	F90 lexer	ObjectZ lexer
Bitmap count	952	738	1025
Bitmap size	10865	3826	2408
Referred pages	2774	1832	1648
Allocated pages	1535	1350	618
Paging gain	73.5%	37.9%	46.4%
Normalization gain	44.6%	26.3%	65.5%
Total gain	85.3%	54.3%	79.9%

The paging gain is computed by comparing the number of non-empty pages with the size of the allocation which would have resulted from a naive implementation based on bitmaps. For instance, when stating that the paging gain for the COBOL lexical analyzer is 73.5%, one must understand that only 26.5% of the pages are non-empty.

The normalization gain is computed *after* the paging gain, that is, it is the gain obtained when comparing the results after paging only, and after paging and normalization. When stating that the normalization gain for the COBOL lexical analyzer is 44.6%, one must understand that the number of unique pages (after normalization) is 55.4% of the total number of pages in all bitmaps.

These results seem to show that this space optimization has more effect on complex lexers, with large bitmaps (COBOL) or many of such bitmaps (ObjectZ).

All in all, the results are pretty good. The true added value of this construction lies in the dictionary and normalization concepts; the properties of the hashing function and the collision handling mechanism (or lack thereof) have shown to be of little practical importance.

It is also very typical that this optimization that was originally designed for resource consumption issues also happened to improve the runtime performance in a noticeable way, despite the added processing. This improvement is mostly due to reduced paging, but also to the ability to compute a bitmap's hash value and to compare bitmaps more efficiently.

7.28 DuraOutput

This module defines the `DuraOutputStream` class that supports automatic identation to ease the generation of meaningful YAFL code.

⟨*Definition* ⟩≡
```
DEFINITION MODULE DuraOutput;

FROM Streams IMPORT OutputStream, VersionedOutputStream;

  CLASS DuraOutputStream;
    INHERITS VersionedOutputStream;

    METHOD SetIndent (Indent: INTEGER);
    METHOD Indent: INTEGER;
    METHOD IncIndent (Increment: INTEGER);
    METHOD DecIndent (Decrement: INTEGER);

    ⟨Facility methods, definition 861b⟩

  END DuraOutputStream;

END DuraOutput;
```

⟨*Implementation* ⟩≡
```
IMPLEMENTATION MODULE DuraOutput;


  CLASS DuraOutputStream;
    INHERITS VersionedOutputStream;

    ⟨Reading and setting the indent level 860a⟩
    ⟨Redefining some of the inherited methods 860b⟩
    ⟨Facility methods, implementation 862⟩

  END DuraOutputStream;

END DuraOutput;
```

860a ⟨*Reading and setting the indent level* 860a⟩≡

```
VAR
  TheIndent: INTEGER;

METHOD SetIndent (Indent: INTEGER);
  BEGIN
  TheIndent ← Indent;
  END SetIndent;

METHOD Indent: INTEGER;
  BEGIN
  RESULT ← TheIndent;
  END Indent;

METHOD IncIndent (Increment: INTEGER);
  BEGIN
  TheIndent ← TheIndent + Increment;
  END IncIndent;

METHOD DecIndent (Decrement: INTEGER);
  BEGIN
  TheIndent ← TheIndent - Decrement;
  END DecIndent;
```

860b ⟨*Redefining some of the inherited methods* 860b⟩≡

```
VAR
  CurColumn: INTEGER;

REDEFINE METHOD WriteLn;
  BEGIN
  BASE;
  CurColumn ← 0;
  END WriteLn;
```

This redefinition of the `WriteSlice` method uses an attribute `Transparent` to prevent it from becoming infinitely recursive, since it calls `WriteChar` and `WriteLn`, which, in turn, call `WriteSlice`.

861a ⟨*Redefining some of the inherited methods* 860b⟩+≡
```
VAR
  Transparent: BOOLEAN;

REDEFINE METHOD WriteSlice (Data: ARRAY OF CHAR;
                            From, Len: INTEGER);
  CONST
    PageWidth = 92;
  BEGIN
  IF Transparent THEN
    BASE (Data, From, Len);
   ELSE
    IF CurColumn = 0 THEN
      IF TheIndent < 0 THEN
        TheIndent ← 0;
        END;
      Transparent ← TRUE;
      FOR i ∈ 1 .. TheIndent DO
        WriteChar (“ ”);
        END;
      Transparent ← FALSE;
      CurColumn ← TheIndent;
     ELSIF CurColumn + Len > PageWidth THEN
      Transparent ← TRUE;
      WriteLn;
      FOR i ∈ 1 .. TheIndent + 2 DO
        WriteChar (“ ”);
        END;
      Transparent ← FALSE;
      CurColumn ← TheIndent + 2;
      END;
    BASE (Data, From, Len);
    END;
  END WriteSlice;
```

861b ⟨*Facility methods, definition* 861b⟩≡
```
METHOD Else;
METHOD End;
```

862 ⟨*Facility methods, implementation* 862⟩≡

```
METHOD Else;
  BEGIN
  DecIndent (1);
  WriteLine (“ELSE”);
  IncIndent (1);
  END Else;

METHOD End;
  BEGIN
  WriteLine (“END;”);
  DecIndent (2);
  END End;
```

7.29 Skeleton files

Skeleton files are used as patterns to generate actual parsers based on a grammatical specification. They are just plain ASCII files, almost just pure YAFL source files, where specific patterns are replaced by parser-specific information.

These patterns are identifiers that start with a dollar sign "$". The patterns that are recognized by the replacement system are listed in table 7.4.

The skeleton files also support a primitive form of conditional compilation, using a `$If` directive, followed by a condition:

```
$If $DynamicVerboseLevel

  METHOD Verbose: BOOLEAN;
  METHOD VerboseLevel: INTEGER;

 $Else

  CONST
    Verbose = $VerboseLevel > 0;
    VerboseLevel = $VerboseLevel;

  $EndIf
METHOD SetVerboseLevel(Level: INTEGER);
```

Just as normal YAFL statements, the conditional compilation `$If` supports an optional `$Else` construct, an requires a matching `$EndIf`. The valid conditions for conditional compilation are listed in table 7.5:

There are of course two skeleton files: one to generate the definition module of the generated parser, and one to generate the corresponding implementation module. See page 274 for more information about how to specify the skeleton to use on the command line when invoking `DURA`.

Table 7.4: Patterns to be replaced in the skeleton file

$CreationMethods	To be replaced by a bunch of methods that are each responsible for creating a new instance of a non-terminal.
$FetchAttributes	To be replaced by a number of `CASE` alternatives that do extract the attributes from an array of `ParserStackElement`, considering a rule number as discriminant expression for the `CASE` statement.
$GenericGotoMethod	For compiled parsers only, is replaced by a number of `CASE` alternatives, that call a rule-specific *goto* method (see $GotoMethods below) depending on the `CASE` statement discriminant expression.
$GenericReduceMethod	To be replaced by `CASE` alternatives, that test for a right-hand side number, and call the proper reduction method.
$GotoMethods	For compiled parsers only, is replaced by a collection of methods, one by rule r, that take a state number s as parameter, and return the state the parser must go to when reducing r starting at state s.
$RootModule	The name of the root module, that is, the module where the root class is defined. The root class is a non-terminal class that serves as start symbol for the generated parser.
$RootClass	The root class (see above).
$ParserModule	The module name of the generated parser.
$ParserClass	The corresponding class name.
$LexerModule	The module of the lexical analyzer. For historical reasons, the lexical analyzer generator `lexyt` generates an analyzer which name matches the enclosing module (For instance, a class `Lexer` defined as part of a module called `Lexer` as well)
$MetaInformationDeclaration	Attribute declarations, that will serve as interfaces to assign values to attributes within non-terminal classes.
$MetaInformationInit	Initialization of the meta-information required to fetch values into non-terminal attributes directly.
$ReduceMethods	To be replaced by the bodies of methods that will implement reductions. One of such methods exists for each right hand side (`FlatRHS`). Since these methods are all parameterless, they must be pasted within the main parse method in order to gain full access to the required environment
$ImportList	To be replaced by an unqualified import list, that is, importing module names instead of importing specific classes. This import list will import all the modules of the project where at least one non terminal class is defined
$StateMachine	To be replaced by the actual push-down automaton.
$RHSToRule	To be replaced by the body of a method which returns a rule number, given a RHS number `RHSNr`.
$MaxStates	To be replaced by the number of states in the PDA
$MaxRHS	To be replaced by the number of right-hand side that have been produced by the original grammar
$MaxRules	To be replaced by the number of flattened grammar rules that have been produced by the original grammar
$TableFileName	In case of interpreted parsers, to be replaced by the name of the file that holds the transition tables.

Table 7.5: Conditions for conditional expansion

$TableDriven	True when the parser we're generating is interpreted.
$CaseDriven	True when the parser we're generating is compiled (generated using YAFL CASE statements).
$DynamicVerboseLevel	True if the level of verbosity can be changed at runtime.

Let's first have a quick look at the definition module skeleton:

865 ⟨*dskel.dura* 865⟩≡

```
DEFINITION MODULE $ParserModule;

IMPORT $RootModule;
IMPORT $LexerModule;

FROM DuraBase IMPORT BaseParser, ParserStatAcc;

  CLASS $ParserClass;
    INHERITS BaseParser ($RootModule.$RootClass, $LexerModule);

    CONST
      DuraVersion = $Version;

    ⟨Handle possibly dynamic verbosity level 866a⟩

    REDEFINE METHOD CREATE;
    REDEFINE METHOD Parse(Lexer: $LexerModule):
           $RootModule.$RootClass;
  END $ParserClass;

END $ParserModule;
```

If the verbosity level is statically defined, we'll take advantage of the YAFL (or ultimately, the `C`) compiler to remove unnecessary code by defining `Verbose` as a constant rather than as a method.

866a ⟨*Handle possibly dynamic verbosity level* 866a⟩≡

```
$If $DynamicVerboseLevel

  METHOD Verbose: BOOLEAN;
  METHOD VerboseLevel: INTEGER;

 $Else

  CONST
    Verbose = $VerboseLevel > 0;
    VerboseLevel = $VerboseLevel;

  $EndIf
```

The following method is always made public to ensure that the interfaces remain the same when one goes from a parser with a dynamic verbosity level to an optimized parser with verbosity level set to 0.

866b ⟨*Handle possibly dynamic verbosity level* 866a⟩+≡

```
METHOD SetVerboseLevel(Level: INTEGER);
```

The verbosity level is an integer which must be understood as a bitmap, where each of the first four bits indicates that a given form of trace must be produced. For instance, if the third bit (starting the bit numbering at 0) is set by something of the form: `SetVerboseLevel(SYSTEM.SetBit(VerboseLevel, 3))`, the entire error recovery process will be traced on the standard output stream.

866c ⟨*Handle possibly dynamic verbosity level* 866a⟩+≡

```
CONST
  LexTrack = 0;
  StepTrack = 1;
  StackTrack = 2;
  ErrorRecoveryTrack = 3;
```

Let's now investigate the implementation module skeleton.

866d ⟨*iskel.dura* 866d⟩≡

```
IMPLEMENTATION MODULE $ParserModule;

⟨Import list 867a⟩

  ⟨The actual parser class 868a⟩

END $ParserModule;
```

Besides importing common classes required for all parsers, the import list for all possible nonterminals must be generated as well by using the `$ImportList` pattern.

867a ⟨*Import list* 867a⟩≡
```
IMPORT ArrayList;
FROM List IMPORT List;
FROM Streams IMPORT StdErr, InputStream, StdOut, OutputStream;
IMPORT Ref;
FROM MetaClass IMPORT DataDictionary, MetaClass, Attribute,
                              ClassAttribute, StringAttribute,
                      IntegerAttribute, BooleanAttribute,
                      AttributeInformation;
IMPORT YLexicalAnalyzer;
FROM DuraBase IMPORT BaseParser, ParserStackElement,
                     ShiftStackElement, ReduceStackElement,
                     BaseNonTerminal;
IMPORT DuraBase;
IMPORT SYSTEM;

⟨Importing IntDictionary if required 867b⟩

$ImportList
```

If we generate interpreted parsers, we must import the `IntDictionary` class (See page 764), that implements space optimization and input/output facilities for large numbers of arrays of integers.

867b ⟨*Importing* `IntDictionary` *if required* 867b⟩≡
```
$If $TableDriven
  FROM DuraIntDictionary IMPORT IntDictionary;
$EndIf
```

868a ⟨*The actual parser class* 868a⟩≡
```
CLASS $ParserClass;
  INHERITS BaseParser ($RootModule.$RootClass, $LexerModule);

  ⟨Attributes and constants 868b⟩

  ⟨Creation method 872a⟩
  ⟨Pasting DURA-generated methods 874a⟩

  ⟨GenericReduce method 875b⟩
  ⟨GenericGoto method 874c⟩
  ⟨Generic Shift method 876a⟩
  ⟨Fold elements from the LALR stack 877b⟩
  ⟨Fetching methods 880⟩
  ⟨Checking whether a given Attribute is fetched by this parser 885b⟩

  $If $CaseDriven

    ⟨Checking whether a given ParserStackElement is null, table-driven 878b⟩

  $Else

    ⟨Checking whether a given ParserStackElement is null, compiled 879a⟩

  $EndIf

  ⟨Backtracking methods 870d⟩
  ⟨Error recovery method 900a⟩
  ⟨Debugging methods 907⟩
  ⟨StatAcc method implementation 911a⟩

  ⟨Main parsing method 890c⟩
  ⟨Consulting the parser's state 890b⟩

  END $ParserClass;
```

`LrStackSize` defines the LR stack size. Since it is used extensively, using a static value rather than reallocating it dynamically will have a significant impact on the performance.

868b ⟨*Attributes and constants* 868b⟩≡
```
CONST
  LrStackSize = 512;
```

If the verbose level ought to be dynamic, we must declare an attribute that holds it.

869a ⟨*Attributes and constants* 868b⟩+≡
```
$If $DynamicVerboseLevel
  VAR
    TheVerboseLevel: INTEGER;
$EndIf
```

`TheStack` implements the LR stack, while `TheSp` implements the top element pointer on this stack. `TheRecoveryStack` and `TheRecoverySp` hold the state of this stack at the most advanced recognized position within the source code. Whenever the parser backtracks (See method *Parser skeleton.Parser skeleton.BackTrack* on page 889), if the currently recognized position is more advanced than the previously most advanced recognized position, `TheRecoveryStack` and `TheRecoverySp` are set to reflect `TheStack` and `TheSp` so that if backtracking happens to fail, one can produce comprehensive error reports and perform an error recovery attempt starting at this most advanced position.

869b ⟨*Attributes and constants* 868b⟩+≡
```
VAR
  TheRecoveryStack,
  TheStack: ARRAY OF ParserStackElement;
  TheRecoverySp,
  TheSp: INTEGER;
```

`TheStateNr` holds the current state number in the PDA. Depending on this state number and on the lookahead token, the various actions can be selected.

869c ⟨*Attributes and constants* 868b⟩+≡
```
TheStateNr: INTEGER;
```

`TheActionNr` holds the action number, in the case where in a given state and with a given lookahead token, several actions (a possible shift and a zero or more reduces) can be taken. When backtracking, `TheActionNr` will be incremented to try the next available action.

869d ⟨*Attributes and constants* 868b⟩+≡
```
TheActionNr: INTEGER;
```

`TheFailure` holds `TRUE` if the parser has reached a dead end, that no action can be taken in the current state and with the current lookahead character, and backtracking must be started.

869e ⟨*Attributes and constants* 868b⟩+≡
```
TheFailure: BOOLEAN;
```

`TheLex` holds the lexer the parser uses to parse the input. This lexer is given as parameter to the `Parse` method (See page 890), but it is more convenient to be set as attribute than passed as parameter to all methods that might potentially use it.

869f ⟨*Attributes and constants* 868b⟩+≡
```
TheLex: $LexerModule;
TheLineCount: INTEGER;
```

The following attributes are only relevant when dealing with interpreted (understand, table-driven) parsers. When dealing with compiled parsers, the equivalent functionality is provided through methods that contain extensive CASE statements.

870a ⟨*Attributes and constants* 868b⟩+≡
```
$If $TableDriven
  TheIntDict: IntDictionary;
  TheArrTable: ARRAY OF ARRAY OF INTEGER;

  TheGotoTable,
  TheLengthTable,
  TheRHSToRuleNrTable,
  TheNullableTable: ARRAY OF INTEGER;

  TheAutomatonTable: ARRAY OF ARRAY OF INTEGER;

  TheRecoverRuleNrTable,
  TheRecoverTable,
  TheRecoverSyncTable: ARRAY OF ARRAY OF INTEGER;
  $EndIf
```

`TheSting` indicates the most advanced position within the input that has been reached. We will assume that a parse operation that fails by backtracking should be due to something that has been found at this `TheSting` position.

870b ⟨*Attributes and constants* 868b⟩+≡
```
TheSting: INTEGER;
```

`TheErrorCount` holds the number of errors that have been encountered so far, and will be compared with `TheErrorLimit` to decide whether it is worth attempting to continue parsing. If `TheErrorLimit` is zero or less, the parser will not stop before the entire input has been parsed.

870c ⟨*Attributes and constants* 868b⟩+≡
```
TheErrorCount,
TheErrorLimit: INTEGER;
```

870d ⟨*Backtracking methods* 870d⟩≡
```
REDEFINE METHOD SetErrorLimit (TheLimit: INTEGER);
  BEGIN
  TheErrorLimit ← TheLimit;
  END SetErrorLimit;
```

`TheStatus` holds the status of the parser, to indicate whether it is currently parsing an input, and to provide information about the reason that caused it to stop. Its possible values are denoted by symbolic constants defined in `BaseParser` (See page 323):

Parsing indicates that the parser is busy parsing an input

StepLimitError indicates that the parser has stopped because it has reached its limit in terms of allowed unsuccessfull backtracking steps

RecoverableError indicates that the parser has terminated parsing, at least one error has been found, but the parser has been able to recover to an apparently sound state

UnrecoverableError indicates that the parser has encountered an error from which it could not recover

NoError indicates that the parser has finished parsing its input successfully.

871a ⟨*Attributes and constants* 868b⟩+≡
```
TheStatus: INTEGER;
```

`TheStepNr` is incremented on each evolution of our LR automaton

`TheStingStepNr` is also incremented on each evolution of the automaton, but reset to 0 whenever the parser advances beyond the previously most advanced recognized position. While `TheStepLimit` holds the maximum number of steps the parser will perform *without advancing on the input*. If `TheStepLimit` is zero or less, the parser won't stop before the entire input has been analyzed.

871b ⟨*Attributes and constants* 868b⟩+≡
```
TheStepNr,
TheStingStepNr,
TheStepLimit: INTEGER;
```

871c ⟨*Attributes and constants* 868b⟩+≡
```
TheGlobalAttrList: ArrayList (Attribute);
```

`TheAdvancing` indicates whether the parser is currently advancing, or backtracking. It is set to `TRUE` by `Shift` (See page 875) and set to `FALSE` by `Back-Track` (See page 889).

871d ⟨*Attributes and constants* 868b⟩+≡
```
TheAdvancing: BOOLEAN;
```

`DURA`-generated parsers must be able to fetch attributes within objects, bypassing YAFL's encapsulation mechanism. This requires access to the data dictionary that describes all classes and attached attributes at runtime.

The `$MetaInformationDeclaration` pattern is replaced by a set of attribute declarations (See method *DuraSpecification.SyntacticSpecification.GenerateMetaInformationDeclaration* on page 307), that will serve as interfaces to assign values to attributes within non-terminal classes.

871e ⟨*Attributes and constants* 868b⟩+≡
```
$MetaInformationDeclaration
```

872a ⟨*Creation method* 872a⟩≡

```
REDEFINE METHOD CREATE;

  METHOD FindAttribute (MClass: MetaClass;
                        Name: ARRAY OF CHAR): Attribute;
    BEGIN
    RESULT ← MClass.FindAttribute (Name);
    IF RESULT = Λ THEN
      StdOut.WriteLine (''Cannot find attribute '' + Name +
                        '' within class '' + MClass.Name);
     ELSE
      TheGlobalAttrList.Append (RESULT);
      END;
    END FindAttribute;

  BEGIN
  BASE;
  TheGlobalAttrList.CREATE;
```

`$MetaInformationInit` is replaced by the code that initializes the meta-information required to fetch values into non-terminal attributes directly (See method *DuraSpecification.SyntacticSpecification.GenerateMetaInformationInit* on page 307).

872b ⟨*Creation method* 872a⟩+≡

```
$MetaInformationInit
```

The arrays that implements the LR stack and the corresponding recovery stack are initialized.

872c ⟨*Creation method* 872a⟩+≡

```
TheStack.CREATE (LrStackSize);
TheRecoveryStack.CREATE (LrStackSize);
```

Then, if we are dealing with an interpreted parser, we load the `IntDictionary` that holds the various table information, and assigns the various attributes that actually implement the transitions and other tabular stuff. Loading the `IntDictionary` happens by calling the `ReadIntDict` method (See page 330).

We make extensive use of the ability offered by `IntDictionary` (See page 764) to associate a name with an entry in the dictionary.

`IntDictionary` uses negative values to refer to arrays. For instance, arrays of arrays are stored as single-dimensioned arrays which entries are all negative. In order to be consistent with this usage, `FindName` returns a negative value when given a valid name, that is, a name that has actually been stored in the dictionary.

873 ⟨*Creation method* 872a⟩+≡

```
$If $TableDriven
  TheIntDict ← ReadIntDict ($TableFileName);
  IF TheIntDict = Λ THEN
    StdErr.WriteLine (“Cannot load table file : ” + $TableFileName);
    ASSERT FALSE;
    END;
  TheArrTable ← TheIntDict.MainArray;
  TheGotoTable ← TheArrTable[-TheIntDict.FindName(“Goto”)];
  TheLengthTable ← TheArrTable[-TheIntDict.FindName(“Length”)];
  TheRHSToRuleNrTable ←
         TheArrTable[-TheIntDict.FindName(“RHSToRuleNr”)];
  TheNullableTable ←
         TheArrTable[-TheIntDict.FindName(“Nullable”)];

  TheAutomatonTable ← TheIntDict.GetArrayArray
                              (TheIntDict.FindName(“Automaton”));
  TheRecoverTable ← TheIntDict.GetArrayArray
                          (TheIntDict.FindName(“Recover”));
  TheRecoverSyncTable ← TheIntDict.GetArrayArray
                          (TheIntDict.FindName(“RecoverSync”));
  TheRecoverRuleNrTable ← TheIntDict.GetArrayArray
                          (TheIntDict.FindName(“RecoverRuleNr”));
$EndIf

  END CREATE;
```

7.29.1 The automaton’s methods

This section describes the various methods that are used by the LR automaton. Some of these methods are *generic* methods, that is, methods that apply in any circumstances. For instance, a generic reduce method is a method that performs *any* reduction, based on a parameter, as opposed to various specific reduction methods, that perform each a single reduction (understand: based on a specific right-hand side).

Well, beloved reader, just in case you wondered what such a specific reduction method might look like, you'll be disappointed. Such methods are generated by DURA, when it encounters the $ReduceMethods pattern in the skeleton file (See method *DuraFlSpec.FlatSpecification.GenerateReduceMethods* on page 582). DURA will replace this pattern by a set of methods, one for every right-hand side (RHS) in the flattened representation. Depending on the nature of this RHS, the corresponding reduction method will be different: attributes will be fetched, lists will be concatenated, etc...

874a ⟨*Pasting* DURA*-generated methods* 874a⟩≡
```
$ReduceMethods
```

The $CreationMethods pattern will be replaced by methods that create new instances of all classes that come into DURA-generated parse trees (See method *DuraSpecification.SyntacticSpecification.GenerateCreationMethods* on page 307).

874b ⟨*Pasting* DURA*-generated methods* 874a⟩+≡
```
$CreationMethods

METHOD PreviousState(Depth: INTEGER): INTEGER;
  BEGIN
  RESULT ← TheStack[TheSp-Depth-1].ToStateNr;
  END PreviousState;
```

GenericGoto implements the *goto* function of an LR automaton as described in . Given a state, after reducing a given rule, determine the state to which the automaton should move.

This method has two versions, one for interpreted parsers, and one for compiled parsers. They are basically the same, except for the fact that the compiled version statically expands the *goto* function into nested CASE statements; while the interpreted version performs the same operations based on arrays that are read into memory at runtime.

874c ⟨GenericGoto *method* 874c⟩≡
```
$If $CaseDriven
  $GotoMethods
  ⟨GenericGoto for compiled parsers 874d⟩
$Else
  ⟨GenericGoto for interpreted parsers 875a⟩
$EndIf
```

874d ⟨GenericGoto *for compiled parsers* 874d⟩≡
```
METHOD GenericGoto (RuleNr, PreviousState: INTEGER): INTEGER;
  BEGIN
  CASE RuleNr OF
    $GenericGotoMethod
    END;
  END GenericGoto;
```

875a ⟨GenericGoto *for interpreted parsers* 875a⟩≡

```
METHOD GenericGoto (RuleNr, PreviousState: INTEGER): INTEGER;
  BEGIN
  RESULT ← TheArrTable[-TheGotoTable_RuleNr] _PreviousState;
  END GenericGoto;
```

Reducing generically is just a matter of calling the proper specific reduction method, depending on the right hand side we wish to reduce.

875b ⟨GenericReduce *method* 875b⟩≡

```
METHOD GenericReduce (RHSNr: INTEGER);
  VAR
    TargetState,
    Len: INTEGER;
  BEGIN
  CASE RHSNr OF
    $GenericReduceMethod
```

When dealing with interpreted parsers, when a reduction basically does nothing meaningful (for instance, when reducing a rule into another based on reverse inheritance), no specific reduction method is generated. A universal scheme is used instead, that reduces the top slice of the proper length from the LR stack.

875c ⟨GenericReduce *method* 875b⟩+≡

```
  $If $TableDriven
 ELSE
  Len ← TheLengthTable_RHSNr;
  TargetState ← GenericGoto (TheRHSToRuleNrTable_RHSNr,
                             PreviousState(Len));
  Λ ← ReduceTopSlice (Len,RHSNr, TargetState);
  $EndIf
  END;
END GenericReduce;
```

In a more common LR automaton, such as generated by `yacc`, for instance, shifting is merely a question of pushing the current symbol onto the stack, reading the next symbol from the input, and setting a new state.

All these operations can be found in this chunk, except for the fact that it gets slightly more complex due to the ability to backtrack, that is, to undo this shift operation, and possibly perform another operation. To achieve this purpose, one cannot just push the symbol on the stack, one must also keep track of the entire parser's state in order to be able to restore it if required.

Hence, the symbol is not pushed on the stack as is, it is first embedded in a `ShiftStackElement` (See page 335) that contains all the state information required to possible undo the shift if required.

876a ⟨*Generic* `Shift` *method* 876a⟩≡
```
METHOD Shift(NewState: INTEGER);
  VAR
    TheElement: ShiftStackElement;
    j: INTEGER;
  BEGIN
  TheShiftCount ← TheShiftCount + 1;
  TheElement.CREATE(FromStateNr ← TheStateNr,
                    ToStateNr ← NewState,
                    ActionNr ← TheActionNr,
                    LexPos ← TheLex.CurrentPosition,
                    LexLength ← TheLex.CurrentLength,
                    LexCarnation ← TheLex.CurrentIncarnation,
                    LexCode ← TheLex.CurrentToken);
  TheStack TheSp ← TheElement;
  TheSp ← TheSp + 1;
  TheLex.Advance;
```

YAFL If the parser we are generating has a statically defined verbose level, `DURA` defines `Verbose` as a boolean constant, and in the case where this statically defined verbose level is 0, the following chunk of code is removed by the YAFL compiler using compile-time constant evaluation.

Otherwise, it produces a trace of the current state of the system somewhere around every 128 characters of the input.

876b ⟨*Generic* `Shift` *method* 876a⟩+≡
```
IF Verbose ∧ SYSTEM.TestBit ($VerboseLevel, LexTrack) THEN
  IF TheLex.CurrentPosition - TheLastPos > 128 THEN
    TheLastPos ← TheLex.CurrentPosition;
    DumpLexStatus (“.”);
    END;
  END;
```

`TheLastPos` holds the position where the last dump of the current state of the parser was performed (see above).

876c ⟨*Attributes and constants* 868b⟩+≡
```
TheLastPos: INTEGER;
```

We now move to the new state. If the current position is more advanced than the previous most advanced position, `TheSting` is updated, and `TheStingStepNr` is reset.

Whenever we shift beyond the previously known most advanced position in the source, we set `TheAdvancing` (See page 871) to `TRUE`.

877a ⟨*Generic* `Shift` *method* 876a⟩+≡

```
⟨Move the parser to state NewState 878a⟩
IF TheLex.CurrentPosition > TheSting THEN
  TheSting ← TheLex.CurrentPosition;
  TheStingStepNr ← 0;
  TheAdvancing ← TRUE;
  END;
END Shift;
```

Similarly to the `Shift` method presented above, the `ReduceTopSlice` method stores the parser's entire context in order to be able to restore it afterwards if required.

`ReduceTopSlice` takes three parameters:

`Depth` is the number of items to reduce (understand: remove) from the LR stack

`RHSNr` specifies the grammar's RHS which is being reduced

`NewState` gives the state the parser must go to after this reduction.

`ReduceTopSlice` returns the `ReduceStackElement` that was pushed on the LR stack as the result of the reduction.

877b ⟨*Fold elements from the LALR stack* 877b⟩≡

```
METHOD ReduceTopSlice (Depth,
                       RHSNr,
                       NewState: INTEGER): ReduceStackElement;
  BEGIN
  TheReduceCountArr[RHSNr] ← TheReduceCountArr[RHSNr] + 1;
  RESULT.CREATE(FromStateNr ← TheStateNr,
                ToStateNr ← NewState,
                ActionNr ← TheActionNr,
                LexPos ← TheLex.CurrentPosition,
                LexLength ← TheLex.CurrentLength,
                LexCarnation ← TheLex.CurrentIncarnation,
                RHSNr,
                TheStack.SLICE (TheSp-Depth, Depth));
  TheSp ← TheSp - Depth;
  TheStack[TheSp] ← RESULT;
  TheSp ← TheSp + 1;
  ⟨Move the parser to state NewState 878a⟩
  END ReduceTopSlice;
```

Moving the parser to a new state happens by setting the attribute `TheStateNr`, and by stating that in this new state, the action to try next has index 0. By backtracking, the other possible actions attached to state `NewState` will be tried successively if necessary.

878a ⟨*Move the parser to state* `NewState` 878a⟩≡

```
TheActionNr ← 0;
TheStateNr ← NewState;
```

When dealing with compiled parsers, a `ParserStackElement` can only be null if the corresponding rule is nullable. This first check is implemented when replacing the `$IsVoidMethod` pattern. All rules that are statically known not to be nullable are not even checked for their content.

878b ⟨*Checking whether a given* `ParserStackElement` *is null, table-driven* 878b⟩≡

```
METHOD IsVoid (p: ParserStackElement): BOOLEAN;
  VAR
    Arr: ARRAY OF ParserStackElement;
  BEGIN
  IF p = Λ THEN
    RESULT ← TRUE;
   ELSE
    WHAT p OF
      IN ReduceStackElement:
        CASE TAG.RHSNr OF
          FinalState:
            END;
          $IsVoidMethod
         ELSE
          END;
        END;
     ELSE
      -- If p is a shift, it is of course not void !!!
      END;
    END;
  END IsVoid;
```

Things are pretty similar when dealing with interpreted parsers, except for the fact that the a priori verification is implemented by `TheNullableTable` (See page 870), so that a reduction of a non-nullable rule is not tested any further.

879a ⟨*Checking whether a given* `ParserStackElement` *is null, compiled* 879a⟩≡

```
METHOD IsVoid (p: ParserStackElement): BOOLEAN;
  VAR
    Arr: ARRAY OF ParserStackElement;
    RHSNr: INTEGER;
  BEGIN
  IF p = Λ THEN
    RESULT ← TRUE;
   ELSE
    WHAT p OF
      IN ReduceStackElement:
        RHSNr ← TAG.RHSNr;
        IF ⟨The reduction is a real one 879b⟩
              ∧ (TheNullableTable_RHSNr = 0) THEN
          IF TheLengthTable_RHSNr = 0 THEN
            RESULT ← TRUE;
           ELSE
            RESULT ← ∀El ∈ TAG.Array • IsVoid(El);
            END;
          END;
        END;
     ELSE
      -- If p is a shift, it is of course not void !!!
      END;
    END;
  END IsVoid;
```

If the reduction is a phony one, created by the error recovery mechanism (See method *Parser skeleton.Parser skeleton.Recover* on page 900) its RHS number is negated. Such a reduction cannot be tested for nullity.

879b ⟨*The reduction is a real one* 879b⟩≡

```
(RHSNr ≥ 0)
```

7.29.2 Fetching methods

The following methods can be used to fetch parts of a parse tree that has been recognized and synthesized in a `ParserStackElement`.

For instance, `ExtractInteger` returns the lexical code attached to the rightmost (understand: the last) lexeme that has been recognized when parsing p. Hence:

```
GRAMMAR
  Op := Expr (Plus OR Minus) Expr;

VAR
  Op: INTEGER;
```

will assign the code attached to the last lexeme of the first sub-expression to `Op`, while

```
GRAMMAR
  Expr Op := (Plus OR Minus) Expr;

VAR
  Op: INTEGER;
```

will assign the operator's code, which is perhaps more likely to be the proper thing to do.

880 ⟨*Fetching methods* 880⟩≡

```
METHOD ExtractInteger (p: ParserStackElement): INTEGER;
  VAR
    Arr: ARRAY OF ParserStackElement;
  BEGIN
  WHAT p OF
    IN ShiftStackElement:
      RESULT ← TAG.LexCode;
      END;
    IN ReduceStackElement:
      Arr ← TAG.Array;
      FOR i ∈ Arr.SIZE - 1 .. 0 BY -1 WHILE RESULT = 0 DO
        RESULT ← ExtractInteger(Arr_i);
        END;
      END;
    END;
  END ExtractInteger;
```

ExtractString returns the image (as a character string) of the rightmost lexeme that was recognized when parsing p.

881a ⟨*Fetching methods* 880⟩+≡

```
METHOD ExtractString (p: ParserStackElement): ARRAY OF CHAR;
  VAR
    Arr: ARRAY OF ParserStackElement;
  BEGIN
  WHAT p OF
    IN ShiftStackElement:
      ⟨p is a ShiftStackElement 881b⟩
      END;
    IN ReduceStackElement:
      ⟨p is a ReduceStackElement 881c⟩
      END;
    END;
  END ExtractString;
```

Extracting the image from a ShiftStackElement is merely a question of asking the lexer for the image of the token at p's position and length.

881b ⟨p *is a* ShiftStackElement 881b⟩≡

```
RESULT ← TheLex.GetCharStream.GetImage (TAG.LexPos,
                                        TAG.LexLength);
```

If p is a ReduceStackElement, we visit the sub-tree recursively from right to left until we find a ShiftStackElement.

881c ⟨p *is a* ReduceStackElement 881c⟩≡

```
Arr ← TAG.Array;
IF Arr = Λ THEN
  DEBUG
   DumpLexStatus (‘‘Extracting string from a void place:”);
    END;
 ELSE
  FOR i ∈ Arr.SIZE - 1 .. 0 BY -1 WHILE RESULT = Λ DO
    RESULT ← ExtractString(Arr_i);
    END;
  END;
```

Unlike the two methods presented above, ExtractNonTerminal truly returns the leftmost non-terminal that has been recognized, *even if void*. ExtractInteger and ExtractString return the code and the image of the first token[17] that has been recognized when parsing p, even if hundreds of void reductions have been necessary before getting as far as recognizing this first (implicitely, non-empty) lexeme.

Also unlike the two previous methods, ExtractNonTerminal is unrecursified because experience has shown that it sometimes required visiting very deep reduction chains.

[17] And there is of course no such thing as an empty token !

`ExtractNonTerminal` returns the leftmost non-terminal, and might return VOID to indicate that this non-terminal was reduced using an empty reduction.

882 ⟨*Fetching methods* 880⟩+≡

```
METHOD ExtractNonTerminal (p: ParserStackElement):
                                            BaseNonTerminal;
  VAR
    q: ParserStackElement;
  BEGIN
  q ← p;
  WHILE (q ≠ Λ) ∧ (RESULT = Λ) DO
    WHAT q OF
      IN ReduceStackElement:
        RESULT ← TAG.NonTerminal;
        IF RESULT = Λ THEN
          IF (TAG.Array ≠ Λ) ∧ (TAG.Array.SIZE > 0) THEN
            q ← TAG.Array0;
           ELSE
            q ← Λ;
            END;
          END;
        END;
     ELSE
      q ← Λ;
      END;
    END;
  END ExtractNonTerminal;
```

`FetchList` fetches a non-terminal and its successors onto a list. Beware of the `WHAT` statement that ultimately performs the type checking when dealing with lists. There is no other check for type consistency between list declarations and usage. For instance, code such as:

```
GRAMMAR
  LoopList := Statement * BY SemiColon;

VAR
  LoopList: NTList(LoopStatement);
```

will be accepted flawlessly by `DURA`, and it is only at runtime, when recognizing statements that are not loop statements, that the system will crash, complaining about a type mismatch.

883a ⟨*Fetching methods* 880⟩+≡

```
METHOD FetchList (l: ArrayList;
                  p: BaseNonTerminal);
  VAR
    q: BaseNonTerminal;
  BEGIN
  q ← p;
  WHILE q ≠ Λ DO
    ⟨Fetch q in l 883b⟩
    q ← q.Previous;
    END;
  ⟨Ajust the list 883c⟩
  END FetchList;
```

This chunk makes use of a special feature of YAFL (which is not found in any other language as far as I know) that allows an element to be tested against a parameter of a parameterized class. For instance, this chunk tests whether `q` is of the type of the elements that can be stored in `l`, and if it is the case, `q` is appended to `l`. YAFL

883b ⟨*Fetch* `q` *in* `l` 883b⟩≡

```
WHAT q OF
  IN l.Element:
    l.Append (TAG);
    END;
 ELSE
  DumpLexStatus (``Cannot fetch to list " + q.ClassName);
  END;
```

The list is built in reverse order, but it is much faster to use `Append` and reverse the list afterwards using `Reverse`, than using `Insert` which is very inefficient when lists are implemented using arrays. YAFL

`Ajust` optimizes the memory storage of the list, since after that all elements have been put into it, the list will not grow anymore, and the space allocated in prevision of further additions in the list is not necessary.

883c ⟨*Ajust the list* 883c⟩≡

```
l.Ajust;
l.Reverse;
```

884a ⟨*Fetching methods* 880⟩+≡
```
METHOD FetchAttributes(Root: ParserStackElement;
                       Nt: BaseNonTerminal);
VAR
  TheArr: ARRAY OF ParserStackElement;
  TheRHSNr: INTEGER;
  p: BaseNonTerminal;
  ToVisit: ParserStackElement;
BEGIN
ToVisit ← Root;
WHILE ToVisit ≠ Λ DO
  WHAT ToVisit OF
    ⟨If ToVisit is a reduce 884b⟩
    ⟨If ToVisit is not a reduce 884c⟩
    END;
```

When fetching attributes, we must mark the `ReduceStackElements` instances to their attached grammar rule. Furthermore, they must be marked using the `SetFetched` method to avoid them from being fetched again when fetching enclosing grammar rules.

884b ⟨*If* `ToVisit` *is a reduce* 884b⟩≡
```
IN ReduceStackElement:
  IF TAG.IsFetched THEN
    TheArr ← Λ;
   ELSE
    TAG.SetFetched;
    TheArr ← TAG.Array;
    TheRHSNr ← TAG.RHSNr;
    END;
  END;
```

The case covered by this chunk is an error and should not occur in practice ! Fortunately, except in early stages of development of DURA, it did not cause any trouble.

884c ⟨*If* `ToVisit` *is not a reduce* 884c⟩≡
```
IN ShiftStackElement:
  DumpLexStatus (‘‘Error while trying to fetch a terminal”);
  StdOut.WriteLine (‘‘Offending token: ” +
           TheLex.GetCharStream.GetImage (TAG.LexPos,
                                          TAG.LexLength));
  TheArr ← Λ;
  END;
```

After marking the `ReduceStackElement` to prevent it from being fetched again, we can now actually fetch the attributes, depending on the rule. This will be achieved by replacement of the `$FetchAttributes` pattern.

Please keep in mind that this method is partially derecursified, by elimination of tail recursion. Each alternative of the `CASE` statement below will fetch all its attributes except the last one by a recursive call to `FetchAttributes`. The last attribute will be left in `ToVisit`, so that the enclosing loop will take it into consideration without requiring an extra recursive call.

885a ⟨*Fetching methods* 880⟩+≡
```
    ToVisit ← Λ;
    IF TheArr ≠ Λ THEN
      CASE TheRHSNr OF
        $FetchAttributes
       ELSE
        END;
      END;
    END;
  END FetchAttributes;
```

`IsFetched` indicates whether an attribute `Attr` as defined by YAFL's data dictionary system is potentially used by `DURA` to fetch information from a parse tree onto a non-terminal. This facility is used when performing generic operations on a parse tree. It uses `TheGlobalAttrList` (See page 871) where `DURA` stores all the attributes it uses for fetching. YAFL

885b ⟨*Checking whether a given* `Attribute` *is fetched by this parser* 885b⟩≡
```
  REDEFINE METHOD IsFetchedAttribute (Attr: Attribute): BOOLEAN;
    BEGIN
    RESULT ← ∃ a ∈ TheGlobalAttrList • a = Attr;
    END IsFetchedAttribute;
```

7.29.3 Backtracking methods

`UndoTop` considers the top element of the parse stack, and undoes it. Several issues are considered: error identification, lexical position, backtracking, unfolding folded reductions.

886a ⟨*Backtracking methods* 870d⟩+≡

```
METHOD UndoTop;
  VAR
    j: INTEGER;
    p: ARRAY OF ParserStackElement;
    Top: ParserStackElement;
  BEGIN
  IF Verbose ∧
        SYSTEM.TestBit ($VerboseLevel, StepTrack) THEN
    StdOut.WriteLine (“Undoing top”);
    END;
  TheSp ← TheSp - 1;
  Top ← TheStack_TheSp;
  ⟨First, resync on the lexical position of the top element 886b⟩
  ⟨Backtrack by using the next available action number from Top 887a⟩
  WHAT Top OF
    IN ReduceStackElement:
      j ← TAG.RHSNr;
      IF j ≥ 0 THEN
        ⟨Update statistic for unreduces 910d⟩
        END;
      ⟨Top is a reduce, unfold it 887b⟩
      END;
    IN ShiftStackElement:
      ⟨Update statistics for unshifts 910e⟩
      END;
   ELSE
    --
    END;
  END UndoTop;
```

The state number is reset from `Top` and `ReSync` (See page 188) is used to restore the lexer at the state it was in when the top element (shift or reduce) was pushed onto the LR stack.

886b ⟨*First, resync on the lexical position of the top element* 886b⟩≡

```
TheStateNr ← Top.FromStateNr;
TheLex.ReSync(Top.LexPos, Top.LexLength, Top.LexCarnation);
```

Since the reduction using `ActionNr` for `Top` did not succeed – otherwise, we would not be busy with backtracking altogether – we must try the next available action. If there is no such thing as a next available action, `BackTrack` will be called again.

887a ⟨*Backtrack by using the next available action number from* `Top` 887a⟩≡

```
TheActionNr ← Top.ActionNr + 1;
```

If a non-terminal is attached to the top-most `ReduceStackElement`, `RollBack` (See page 347) is called to allow the user to capture the unreduce event, and take any appropriate action.

887b ⟨`Top` *is a reduce, unfold it* 887b⟩≡

```
IF TAG.NonTerminal ≠ Λ THEN
  TAG.NonTerminal.RollBack;
  END;
```

Finally, we actually copy the content of the reduction as a slice of the original LR stack onto this very LR stack.

887c ⟨`Top` *is a reduce, unfold it* 887b⟩+≡

```
p ← TAG.Array;
IF p = Λ THEN
  UndoTop;
 ELSE
  FOR i ∈ 0 .. p.SIZE - 1 DO
    TheStack[TheSp + i] ← p_i;
    END;
  TheSp ← TheSp + p.SIZE;
  END;
```

`LexBackTrack` actually performs lexical backtracking. It should not be called directly: `BackTrack` (See the facing page) should be used instead, to ensure that the proper kind of backtracking is used according to the current state of the parser.

888a ⟨*Backtracking methods* 870d⟩+≡
```
METHOD LexBackTrack;
  BEGIN
  IF Verbose ∧ SYSTEM.TestBit ($VerboseLevel, StepTrack) THEN
    DumpLexStatus (‘‘LexBackTrack before”);
    END;
  ⟨Actually backtrack !!! 888b⟩
  ASSERT TheLex.CurrentToken ≥ 0;
  IF Verbose ∧ SYSTEM.TestBit ($VerboseLevel, StepTrack) THEN
    DumpLexStatus (‘‘LexBackTrack after”);
    END;
  IF TheLex.CurrentToken ≠ TheLex.ERROR THEN
    ⟨Update lexical backtracking statistics 910f⟩
    TheActionNr ← 0;
   ELSE
    UndoTop;
    END;
  END LexBackTrack;
```

Lexical backtracking is implemented by the `BackTrack` method (See page 224).

888b ⟨*Actually backtrack !!!* 888b⟩≡
```
TheLex.BackTrack;
```

`BackTrack` is a wrapper method, that considers the possibility of performing a lexical backtracking (that is, reconsidering whether the current token can be interpreted in a different way). If no such interpretation exists, or if the lexical analyzer simply does not support such level of backtracking, `UndoTop` is called to perform syntactical backtracking.

889a ⟨*Backtracking methods* 870d⟩+≡
```
METHOD BackTrack;
  VAR
    j: INTEGER;
  BEGIN
  ⟨Consider reversal of direction 889b⟩
  WHAT TheStack[TheSp-1] OF
    IN ShiftStackElement:
      LexBackTrack;
      END;
    IN ReduceStackElement:
      ⟨First, consider the reduction cut's level 890a⟩
       ELSE
        UndoTop;
        END;
      END;
    END;
  END BackTrack;
```

If we were advancing (See page 871), and we now backtrack, we must store the current state for possible later recovery. This operation is of course not repeated if several negative operations (understand: unshifts or unreduces) are performed in a row.

889b ⟨*Consider reversal of direction* 889b⟩≡
```
IF TheAdvancing THEN
  TheAdvancing ← FALSE;
  TheRecoverySp ← TheSp;
  ⟨Copy TheStack onto TheRecoveryStack for later recovery 900c⟩
  END;
```

Before undoing a reduction, we must first consider its cut level. If the reduction is attached to a full-cut grammar rule, attempting to undo it will cause the parser to stop. If it is attached to a snip-cut grammar rule, things are slightly more complex.

It is of course tempting to bluntly remove whatever was related to the reduction to undo from the LR stack if this reduction has been snipped or cut. However, things are not that simple.

If we simply remove the reduction altogether, we might miss an alternate interpretation of the input which would be based on another interpretation of the first token of the faulty reduction. In order to cope with this issue, we simply perform lexical backtracking, but we make sure that undoing the reduce (by means of the `UndoTop` method (See page 886), after that lexical backtracking did not yield anything significant) will not consider any alternate path before the entire rule has been undone, by setting its reduce array to `VOID`. For plain grammar rules that are not marked with any cut-related information, this reduce array must be maintained, because it might be pushed back onto the LR stack when undoing a corresponding reduction. If this reduce array is set to `VOID`, there is no way it can be pushed back onto the LR stack afterwards.

Setting this reduce array to void is not visible in this skeleton, because it is performed by automatically generated code when performing the replacement of the `$ReduceMethods` pattern (See method *DuraReductions.FetchNonTerminalReduction.GenerateReductionCode* on page 828).

890a ⟨*First, consider the reduction cut's level* 890a⟩≡

```
CASE TAG.CutLevel OF
  SnipCut:
    LexBackTrack;
    END;
  FullCut:
    IF Verbose THEN
      StdOut.WriteString (“Stopping on cut: ”);
      IF TAG.NonTerminal ≠ Λ THEN
        StdOut.WriteString (TAG.NonTerminal.ClassName);
        END;
      StdOut.WriteLn;
      END;
    TheStateNr ← ErrorState;
    END;
```

890b ⟨*Consulting the parser's state* 890b⟩≡

```
REDEFINE METHOD Status: INTEGER;
  BEGIN
  RESULT ← TheStatus;
  END Status;
```

Here comes the main parsing method. It initializes the LR stack, pushes the initial state onto it, and iterates, moves the parser's state from state to state, considers backtracking if appropriate, and continues until the final state is reached, in which case a complete parse tree is built and returned, or until the parser fails in recognizing a valid input.

890c ⟨*Main parsing method* 890c⟩≡

```
REDEFINE METHOD Parse(Lexer: $LexerModule):
                            $RootModule.$RootClass;
  VAR
    GoOn: BOOLEAN;
    StackBottom: ShiftStackElement;
    p, q: ARRAY OF INTEGER;
    Nt: BaseNonTerminal;

  $If $TableDriven
    Code, i, j, Cur, Target: INTEGER;
    ActionArray: ARRAY OF INTEGER;
  $EndIf
```

`ErrorState` is the state the parser goes in when a fatal error condition has been detected even after backtracking, and error recovery must be considerd, hoping to get back onto a more state where parsing can resume successfully.

891 ⟨*Attributes and constants* 868b⟩+≡

```
CONST
  ErrorState = -8712;
```

892 ⟨*Main parsing method* 890c⟩+≡

```
BEGIN
⟨Resetting metrics information 910c⟩
⟨Initialize the whole stuff 893a⟩
GoOn ← TRUE;
WHILE GoOn DO
  IF Verbose ∧ SYSTEM.TestBit ($VerboseLevel, StackTrack) THEN
    ⟨Dump current parsing state 893b⟩
    END;
  TheStingStepNr ← TheStingStepNr + 1;
  TheStepNr ← TheStepNr + 1;
  ⟨Check whether the maximum number of steps has been reached 894b⟩
  CASE TheStateNr OF
    FinalState:
      GoOn ← FALSE;
      END;
    ErrorState:
      ⟨In ErrorState, considering recovery 894d⟩
      END;
$If $CaseDriven
    $StateMachine
$EndIf
   ELSE
$If $TableDriven
    ⟨Generate code to interpret TheAutomatonTable 895b⟩
$Else
    ASSERT FALSE;   -- Just in case....
$EndIf
    END;
  IF TheFailure THEN
    BackTrack;
    TheFailure ← FALSE;
    END;
  END;
⟨Finally extract RESULT 897⟩
FOR Idx ∈ 0 .. TheStack.SIZE - 1 DO
  TheStack_Idx ← Λ;
  END;
FOR Idx ∈ 0 .. TheRecoveryStack.SIZE - 1 DO
  TheRecoveryStack_Idx ← Λ;
  END;
TheLex ← Λ;
END Parse;
```

We first create the initial state (as a phony `ShiftStackElement`), which is pushed onto the LR stack. Then, we reset several counters and variables, since the same parser object can be used to parse several inputs in a row.

893a ⟨*Initialize the whole stuff* 893a⟩≡

```
StackBottom.CREATE(ErrorState, 0, 0, 0, 0, 0, 0);
TheStack0 ← StackBottom;
TheLex ← Lexer;
TheLineCount ← TheLex.LineCount;
TheLex.Advance;
TheStateNr ← 0;
TheActionNr ← 0;
TheSting ← 0;
TheErrorCount ← 0;
TheFailure ← FALSE;
TheSp ← 1;
TheStepNr ← 0;
TheStingStepNr ← 0;
TheStatus ← Parsing;
TheAdvancing ← TRUE;
```

There is not much to say about the following chunk, except perhaps that it is rather boring reading.

893b ⟨*Dump current parsing state* 893b⟩≡

```
StdOut.WriteLn;
DumpStack;
StdOut.WriteLn;
StdOut.WriteString (“Step: ”);
StdOut.WriteInt (TheStepNr, 0);
StdOut.WriteString (“.”);
StdOut.WriteInt (TheStingStepNr, 0);
StdOut.WriteString (“ State: ”);
StdOut.WriteInt (TheStateNr, 0);
StdOut.WriteString (“ ActionNr: ”);
StdOut.WriteInt (TheActionNr, 0);
StdOut.WriteString (“ Code: ” +
                    TheLex.TokenImage (TheLex.CurrentToken));
StdOut.WriteString (“ Image: ”);
StdOut.WriteString (TheLex.Image);
StdOut.WriteString (“ LexPos: ”);
StdOut.WriteInt (TheLex.CurrentPosition, 0);
StdOut.WriteString (“ Sp: ”);
StdOut.WriteInt (TheSp, 0);
StdOut.WriteLn;
```

894a ⟨*Backtracking methods* 870d⟩+≡

```
REDEFINE METHOD SetStepLimit (TheLimit: INTEGER);
  BEGIN
  TheStepLimit ← TheLimit;
  END SetStepLimit;
```

894b ⟨*Check whether the maximum number of steps has been reached* 894b⟩≡

```
IF (TheStepLimit > 0) ∧
       (TheStingStepNr ≥ TheStepLimit) THEN
  StdOut.WriteLine (‘‘Error on number of steps. ” +
                    ‘‘Sting is:”);
  ⟨Restore most advanced state, and dump 894c⟩
  TheStatus ← StepLimitError;
  GoOn ← FALSE;
  END;
```

894c ⟨*Restore most advanced state, and dump* 894c⟩≡

```
TheLex.SetCurrentPosition (TheSting);
TheLex.GetCharStream.ComputeLineAndColumnNr
                            (SYSTEM.CHR(SYSTEM.NewLine));
TheLex.ReadCurrentToken;
DumpLexStatus (‘‘Error found:”);
StdOut.WriteString (‘‘ LineNr: ”);
StdOut.WriteInt (TheLex.GetCharStream.LineNr+1, 0);
StdOut.WriteString (‘‘ ColNr: ”);
StdOut.WriteInt (TheLex.GetCharStream.ColumnNr+1, 0);
StdOut.WriteLn;
```

If we are in `ErrorState`, we first reset the LR stack in the state it was in at the most advanced position in the input, which is implemented by `TheSting`.

894d ⟨*In* `ErrorState`*, considering recovery* 894d⟩≡

```
⟨Restore most advanced state, and dump 894c⟩
```

We then call the `Recover` method (See page 900).

895a ⟨*In* `ErrorState`, *considering recovery* 894d⟩+≡

```
TheErrorCount ← TheErrorCount + 1;
IF (TheErrorLimit > 0) ∧
        (TheErrorCount ≥ TheErrorLimit) THEN
  GoOn ← FALSE;
 ELSE
  GoOn ← Recover;
  END;
IF GoOn THEN
  IF TheStatus = Parsing THEN
    TheStatus ← RecoverableError;
    END;
 ELSE
  StdOut.WriteLine (“Error recovery failed”);
  TheStatus ← UnrecoverableError;
  END;
TheFailure ← FALSE;
```

`p` holds the array that describes the actions that must be taken when the parser is in state `TheStateNr`.

895b ⟨*Generate code to interpret* `TheAutomatonTable` 895b⟩≡

```
p ← TheAutomatonTable_TheStateNr;
Code ← TheLex.CurrentToken;
ActionArray ← TheArrTable[−p_Code];
IF ActionArray = Λ THEN
  TheFailure ← TRUE;
 ELSE
  ⟨Perform action extracted of ActionArray 896⟩
  END;
```

`ActionArray` holds the list of applicable actions in a given state, indexed by `TheActionNr`. When backtracking, `TheActionNr` is incremented. When it outgrows the `ActionArray`'s limits, the system moves the `ErrorState`, thereby enabling backtracking.

`ActionArray`'s values are negative to indicate a Shift operation and positive for a Reduce operation (See method *DuraPDA.PushDownAutomaton.StoreArray* on page 656).

896 ⟨*Perform action extracted of* `ActionArray` 896⟩≡

```
IF TheActionNr ≥ ActionArray.SIZE THEN
  TheFailure ← TRUE;
  IF Verbose ∧
       SYSTEM.TestBit ($VerboseLevel, StepTrack) THEN
    StdOut.WriteString (“Cannot perform action: ”);
    StdOut.WriteInt (TheActionNr, 0);
    StdOut.WriteString (“ in state ”);
    StdOut.WriteInt(TheStateNr, 0);
    StdOut.WriteString (“ code is ”);
    StdOut.WriteInt(Code, 0);
    StdOut.WriteString (“ the array is: ”);
    FOR x ∈ ActionArray DO
      StdOut.WriteString(“ ”);
      StdOut.WriteInt (x, 0);
      END;
    StdOut.WriteLn;
    END;
 ELSE
  IF Verbose ∧
       SYSTEM.TestBit ($VerboseLevel, StepTrack) THEN
    StdOut.WriteString (“Performing action: ”);
    StdOut.WriteInt (TheActionNr, 0);
    StdOut.WriteString (“ in state ”);
    StdOut.WriteInt(TheStateNr, 0);
    StdOut.WriteString (“ code is ”);
    StdOut.WriteInt(Code, 0);
    StdOut.WriteString (“ the array is: ”);
    FOR x ∈ ActionArray DO
      StdOut.WriteString(“ ”);
      StdOut.WriteInt (x, 0);
      END;
    StdOut.WriteLn;
    END;
  IF ActionArray_TheActionNr < 0 THEN
    Shift(-ActionArray_TheActionNr);
   ELSE
    GenericReduce (ActionArray_TheActionNr);
    END;
  END;
```

Well, when we have reduced the goal non-terminal of our grammar, it is available on the LR stack, on index 1, since index 0 is used by the phony initial state which was pushed when initializing the stack.

The resulting non-terminal is snipped (See method *DuraBase.BaseNonTerminal. Snip* on page 355) since maintaining any form of backtracking information is now totally superfluous.

897 ⟨*Finally extract* RESULT 897⟩≡

```
IF TheStateNr ≠ ErrorState THEN
  Nt ← ExtractNonTerminal(TheStack₁);
  IF Nt ≠ Λ THEN
    WHAT Nt OF
      IN $RootModule.$RootClass:
        RESULT ← TAG;
        RESULT.Snip;
        END;
     ELSE
       -- No abort, even if this should never happen !
      END;
    END;
  IF TheErrorCount = 0 THEN
    TheStatus ← NoError;
   ELSE
    ASSERT TheStatus ≠ Parsing;
    END;
  END;
```

7.29.4 Error recovery

Error recovery is crucial issue in parser generation. Many users have complained for years about the weaknesses of yacc's error recovery scheme. There are still some major efforts in this direction, some of which use global metrics to minimize the risk of cascading errors [89] as a consequence of dubious error recovery heuristics. Quite unfortunately, these highly sophisticated techniques do not apply to DURA due to their strong bound with the common model of separation between lexical and syntactical analysis, together with its immediate corrolary, namely, lexical inambiguity.

Even if not entirely relevant to the syntactical issue, [90] proposes a framework and methodology to handle cascading error messages using higher order functions. Quite clearly, we must try to perform error handling in the most generic way possible.

The issue can be stated as follows:

- The input is incorrect, that is, the input string is not a possible production of the root symbol. This *global failure* should not be confused with backtracking, which is a *local failure*. In other words, error recovery is the technique that must be applied *after* that trying all possible alternatives using backtracking has shown that it was not a question of trying another early search path. There simply is no way the grammar can match the input.

- We assume that the problem starts at the farthest point in the input that has been reached so far. In other words, when considering error recovery, we'll always look forwards, not backwards. In theory, this might lead to sub-optimal error recovery. In practice, it never did so far. Given the very nature of error recovery, namely, the ability for the parser to continue after that the first error has been discovered, a sub-optimal scheme which is easy to implement might as well be preferred to a slightly more sophisticated construct that would require significantly more work. We don't need safety nests for highly unlikely worst cases. Our error recovery scheme is very naive, and provides good results in most realistic cases.

- We will not consider error repair, that aims at making the parser fix the error automatically, and terminate gracefully, possibly enabling the further passes of compilation. This is only possible if the repair happens to be mundane by nature: inserting a semicolon, ignoring a spurious token. There is no reasonably reliable way a compiler can insert a random identifier out of the blue without jeopardizing the parser's behaviour. Whenever an error has been encountered, error recovery will attempt to proceed with parsing, but no guarantee is given about the validity of resulting parse trees.

- Some sophisticated error-recovery algorithms support three operations, namely token deletion, insertion and replacement. Due to the degree of liberty induced by lexical backtracking, existing strategies based on some form of local or global [89] metrics for the selection of the optimal operation to apply would explode exponentially. One must keep in mind that in the common model, the input is a single list of tokens $x_1, x_2, \dots x_3$, while DURA's model supports multiple incarnations for every token, hence, a number of possible inputs that grows exponentially with the input's size.

Given this extreme degree of liberty, error recovery in DURA is straightforward. The user identifies some of the grammar's non-terminals as requiring automatic error recovery (See page 272). When an error occurs, the parser skips tokens until it encounters a token that can validly follow one of such recoverable non-terminals which is currently being recognized.

In other words, we have:

$$Recoverable \in \mathbb{P}\,\rho FlatRHS$$

$Recoverable$ is a subset of all possible right-hand sides (See page 511) in our grammar, for which the user has explicitely indicated that automatic error recovery must be enabled.

For instance, in `COBOL`, if one marks the non-terminal that represents paragraphs as requiring automatic error recovery, and if an error occurs while recognizing a paragraph, the parser will skip tokens until encountering a token that can validly follow a `COBOL` paragraph. Then, a phony reduction is performed, a part of the LR stack is replaced by a phony non-terminal, and the parser resumes.

This scheme is very naive, but provides reasonable results in most circumstances. It can be improved, by reversing the problem, and instead of considering all tokens that can follow a subset of the non-terminals, one can consider each token, and take the most *discriminant* action. For instance, if a given token cannot occur without following a given non-terminal, we can safely use its presence as a trigger for some form of brute force error recovery. Similarly, if a given token cannot occur in any other place than as terminating a non-terminal, a similar action can be taken.

Depending on the language, or even, depending on the grammar rule within a single grammar the first form based on the $FOLLOW$ set is more appropriate, while in other cases, the second form, based on the $LAST$ set, performs better.

For instance, `FORTRAN` (in its latest version, namely `FORTRAN90`), accepts closed syntactical structures, where `IF` ends with an `ENDIF`, `FUNCTION` with an `ENDFUNCTION`, etc . . . .

Z and ObjectZ are even better examples of this. Most of its constructions are made of LaTeXenvironments, with matching `\begin{...}` and `\end{...}` commands. It is then tempting to claim that if an error has occurred after that a `\begin{...}` command has been encountered, one can synchronize on the matching `\end{...}`. If, at a given nesting level, the error is precisely the absence of the matching `\end{...}`, the parser can synchronize when matching the `\end{...}` at the enclosing nesting level.

These ideas about more advanced error recovery schemes served as motivation for the computation of sets beyond the usual $FIRST$ and $FOLLOW$, namely the $FIRST$, $MIDDLE$, and $PRECEDE$ sets. However, the scheme which is currently implemented is the good old $FOLLOW$ set-based panic scheme.

Error recovery as provided by `DURA` as it stands now has provided acceptable results for rather flat languages, such as `COBOL` while one can expect it to perform poorly on highly recursive and nested languages, such as Pascal and derivatives; where one might resynchronize with the wrong level, generating a cascade of errors later.

900a ⟨*Error recovery method* 900a⟩≡
```
METHOD Recover: BOOLEAN;
  VAR
    Good: BOOLEAN;
    ReduceEl: ReduceStackElement;
    k,
    SyncStateNr, StateNr, NewStateNr,
    SkippedTokens: INTEGER;
    p: ARRAY OF INTEGER;
  BEGIN
  IF TheRecoverTable = Λ THEN

    $If $CaseDriven

      TheRecoverTable.CREATE(TheLex.LASTTOKEN + 1);
      TheRecoverRuleNrTable.CREATE(TheLex.LASTTOKEN + 1);
      $InitRecoverTable
     $Else
      ⟨Error recovery for compiled parsers 900b⟩
      $EndIf
    END;
  ⟨First, restore the most advanced state that has been recognized 901⟩
  IF Verbose THEN
    StdOut.WriteLine (“Error recovery in action...”);
    DumpStack;
    DumpRecoverTable (StdOut);
    END;
```

Error recovery for compiled parsers is not implemented yet.

900b ⟨*Error recovery for compiled parsers* 900b⟩≡
```
ASSERT FALSE;
```

Since DURA generates backtracking parsers, a failure is detected when the input pointer is at the beginning of the input, and that all search paths have been investigated, or if a cut non-terminal has been encountered for undoing. In both cases, the LR stack might have been altered since the true problem occurred.

Therefore, we must first restore the parser in its most advanced state (understand: in its state when it reached its farthest point in the input).

This is why TheStack is copied onto TheRecoveryStack, as well as the corresponding stack pointers when advancing...

900c ⟨*Copy* TheStack *onto* TheRecoveryStack *for later recovery* 900c⟩≡
```
j ← TheSp;
WHILE (j ≥ 0) ∧ (TheRecoveryStack_j ≠ TheStack_j) DO
  TheRecoveryStack_j ← TheStack_j;
  j ← j - 1;
  END;
```

and the `TheStack` is restored before attempting error recovery...

901 ⟨*First, restore the most advanced state that has been recognized* 901⟩≡

```
TheSp ← TheRecoverySp;
FOR i ∈ 0 .. TheSp DO
  TheStack_i ← TheRecoveryStack_i;
  END;
```

This is the error recovery's main loop: skip tokens until a possible phony reduction can reduce one of the right hand sides marked as supporting automatic error recovery (See page 898).

`TheRecoverTable` is a table that, given a token t, returns all the states that can begin the analysis of a rule for which automatic recovery is enabled, and of which t happens to be in the $FOLLOW$ set.

TheRecoverTable, as defined below, gives the states that can cause error recovery to start if found on the LR stack. It is defined formally as taking a *TokenCode* (See page 14) as parameter – which is expected, of course – as well as a grammar in the form of a set of *FlatRule*s (See page 477). This is because the $FOLLOW$ sets cannot be computed for a grammar rule in isolation, without its surrounding grammar.

For the rest, *TheRecoverTable* basically returns the states that contain the first item (See page 623) of right hand sides (See page 511) in *Recoverable* (See page 898) and for which the enclosing grammar rule's $FOLLOW$ set includes the current lookahead character.

$$
\begin{array}{|l}
\mathit{TheRecoverTable} : (\mathbb{P}\,\mathit{FlatRule} \times \mathit{TokenCode}) \mapsto \\
\qquad \mathbb{P}\,\mathit{PDAState} \text{ (See page 638)} \\
\hline
\forall\, t \in \mathit{TokenCode}, g \in \mathbb{P}\,\mathit{FlatRule} \bullet \\
\quad \forall\, s \in \mathit{TheRecoverTable}(g, t) \bullet \\
\quad\quad \exists\, \mathit{RHS} \in \mathit{Recoverable} \bullet \\
\quad\quad\quad \delta \mathit{RHS}.\mathit{FirstItem} \in s \wedge \\
\quad\quad\quad\quad t \in \delta \mathit{RHS}.\mathit{Enclosing}.\mathit{FollowSet}(g)
\end{array}
$$

If t is not in the $FOLLOW$ set of any recoverable token computed using grammar g, $\mathit{TheRecoverTable}(g, t)$ is an empty set, implemented as a `VOID` array.

For instance, if the grammar rule for the `IfStatement` is marked as supporting automatic error recovery, all its right hand sides are in *Recoverable* (See page 898), and $\mathit{TheRecoverableTable}(g, t)$ will contain (but not necessarily exclusively) all states that contain the first item of any of `IfStatement`'s right hand sides if and only if t is in `IfStatement`'s $FOLLOW$ set.

The following loops skips tokens until an error-repair reduction can be performed using one of the states that had been pushed on the LR stack before.

902a ⟨*Error recovery method* 900a⟩+≡

```
WHILE ¬ Good DO
  p ← TheRecoverTable_{TheLex.CurrentToken};
  IF (p ≠ Λ) ∧ (p.SIZE > 0) THEN
    ⟨Considering recovering on the current token 902b⟩
    END;
  IF ¬ Good THEN
    IF TheLex.CurrentToken = TheLex.EOF THEN
      Good ← TRUE;
     ELSE
      ⟨Lexical backtrack, or advance 905d⟩
      END;
    END;
  END;
END Recover;
```

Given p, we now scan the LR stack from bottom to top, trying to perform the shortest phony recovery reduction. For each level on the stack, we keep StateNr and SyncStateNr, and we try to find a rule that can be reduced starting at StateNr, continuing on SyncStateNr.

902b ⟨*Considering recovering on the current token* 902b⟩≡

```
FOR i ∈ TheSp .. 1 BY - 1 WHILE ¬ RESULT DO
  IF (TheStack_{i-1} ≠ Λ) ∧ (TheStack_i ≠ Λ) THEN
    StateNr ← TheStack_{i-1}.ToStateNr;
    SyncStateNr ← TheStack_i.ToStateNr;
    k ← FIRST x ∈ 0 .. p.SIZE - 1 • StateNr = p_x;
    IF k ≥ 0 THEN
      ⟨Consider reduce starting at StateNr, synchronizing on SyncStateNr 904⟩
      END;
    END;
  END;
```

At first sight, it would have been sufficient to test on StateNr to determine whether one can perform a phony reduction here. As a matter of fact, this is precisely the way the first version of this error recovery algorithm worked. Or, stated more accurately, this was the way the first version of this algorithm tried to work. Actually, it did not.

There is a flaw, that makes this simple approach unapplicable. When a state is found on the LR stack, it means that at some point, we have been starting analyzing a given rule. The problem arises when, after that the rule has been analyzed entirely, the state that identifies its start still lies on the LR stack !

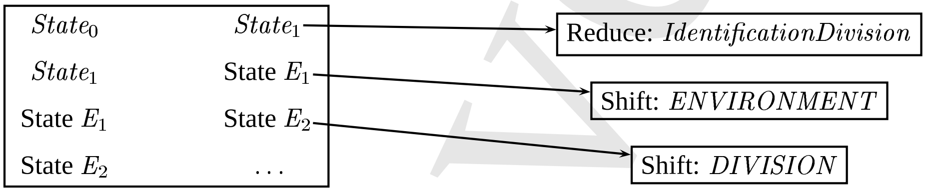
How does this apparent nonsense translates in practice ?

Let's consider a `COBOL` grammar:

$$
\begin{aligned}
\mathit{Program} \ ::= \ & I_0 \ \mathit{IdentificationDivision} \\
& I_1 \ \mathit{EnvironmentDivision} \\
& I_2 \ \mathit{DataDivision} \\
& I_3 \ \mathit{ProcedureDivision} \ I_4
\end{aligned}
$$

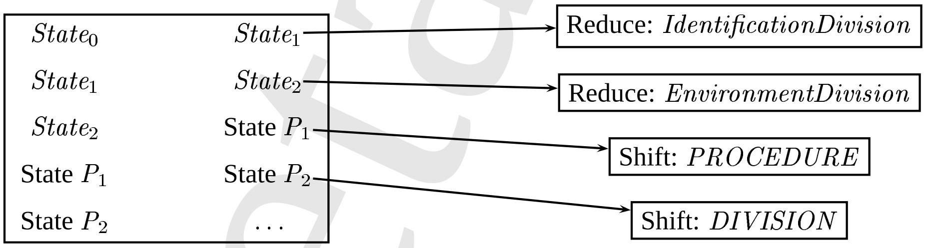
where the I_i's are LR items (See page 623), defining *positions* in a grammar rule including a position before the first symbol (I_0), and a position after the last symbol of the rule (I_4). Let's assume that $\mathit{EnvironmentDivision.RHS} \subset \mathit{Recoverable}$ (See page 898), thereby enabling automatic error recovery for all right hand sides attached to the grammar rule $\mathit{EnvironmentDivision}$. During the recognition of such an $\mathit{EnvironmentDivision}$, the LR stack looks as depicted in figure 7.19. State_0 includes item I_0 as described in the grammar rule above, while State_1 includes item I_1. State_1 identifies the start of the recognition of $\mathit{EnvironmentDivision}$.

Figure 7.19 LR stack when recognizing $\mathit{EnvironmentDivision}$



Then, when analyzing the $\mathit{ProcedureDivision}$, the state of the LR stack is as depicted in figure 7.20. State_1 is still visible on the stack.

Figure 7.20 LR stack when recognizing $\mathit{ProcedureDivision}$



If the parser fails to parse the input, it will consider lookahead tokens starting at the most advanced position in the input and try to perform a dummy reduction of one of the grammar rules it was currently analyzing, hoping to get back onto a more stable state and resume parsing. If a token we are visiting when attempting to recover from a mistake that occured while parsing the $ProcedureDivision$ happens to be an element of $FOLLOW(EnvironmentDivision)$, if based only on the occurrence of $State_1$ on the LR stack, error recovery will reduce a phony $EnvironmentDivision$ while it clearly make no syntacical sense because the $EnvironmentDivision$ the parser started to recognized in $State_1$ has already been entirely recognized.

In order to avoid such problems, DURA also checks the state *after* the state that identifies the beginning of the rule, to ensure that it describes a rule which is currently being recognized, as opposed to a rule that has already been recognized entirely.

This is implemented by $TheRecoverSyncTable$ defined as:

$$TheRecoverSyncTable : (\mathbb{P}\, FlatRule \times PDAState) \mapsto \mathbb{P}\, PDAState \text{ (See page 638)}$$

$$\begin{array}{l}
\forall\, Grammar \in \mathbb{P}\, FlatRule, s_1 \in PDAState \bullet s_2 \in TheRecoverSyncTable(Grammar, s_1) \Leftrightarrow \\
\quad \exists\, i_2 \in s_2.Items \bullet \\
\quad\quad \exists\, i_1 \in s_1.Items \bullet \\
\quad\quad\quad \exists\, r \in Grammar \bullet \\
\quad\quad\quad\quad \exists\, Rhs \in r.RHS \bullet \\
\quad\quad\quad\quad\quad Rhs.Length > 0 \land \\
\quad\quad\quad\quad\quad i_1 = head\ Rhs.Items \land i_2 = head\ tail\ Rhs.Items
\end{array}$$

$tail$ returns a list built by removing the first element of the list it is applied to. $head$ returns the first element of the list it is applied to. Consequently, $head\ tail$ returns the *second* element of the list it is applied to.

$$\begin{array}{l}
\forall\, s \in PDAState \bullet \nexists\, t \in TokenCode \bullet s \in TheRecoverTable(Grammar, t) \Rightarrow \\
\quad TheRecoverSyncTable(Grammar, s) = \phi
\end{array}$$

In other words, given a grammar g, $TheRecoverSyncTable(g, s)$ contains the set of states that can follow state s by one operation (shift of reduce). Hence, if s as a state can start the recognition of a right hand side $r \in Recoverable$, and if the next state on the LR stack s' is such that $s' \in TheRecoverSyncTable(g, s)$, one can safely assume that r is currently being recognized and that its recognition is not terminated. Otherwise, the next state on the LR stack would not be part of $TheRecoverSyncTable(g, s)$. For instance, state E_1 as depicted in figure 7.19 is an element of $TheRecoverySyncTable(State_1)$.

If r is currently being recognized, and $r \in Recoverable$, we can recover from our errors by reducing the top slice of the LR stack, and consider it as a r.

904 ⟨*Consider reduce starting at* StateNr, *synchronizing on* SyncStateNr 904⟩≡

```
IF ∃x ∈ TheRecoverSyncTable_StateNr • x = SyncStateNr THEN
  ⟨Build a phony reduction 905a⟩
  ASSERT Good;
  ASSERT RESULT;
  END;
```

We're almost there: `TheRecoverRuleNrTable` is an array that allows us to retrieve the rule number (or more accurately, the first of the possible rule numbers) for which we have determined that `StateNr` (or `p[k]`) can be a start state.

This rule number is fed to $Goto$ (See page 874) to compute the state of the system after the phony reduction.

905a ⟨*Build a phony reduction* 905a⟩≡
```
IF Verbose THEN
  StdOut.WriteLine (“Matching found”);
  END;
NewStateNr ← GenericGoto
         (TheRecoverRuleNrTable_{TheLex.CurrentToken,k},
          p_k);
```

We create a `ReduceElement`, where the original state number is negated to indicate that it is the result of an error recovery phony reduction.

905b ⟨*Build a phony reduction* 905a⟩+≡
```
ReduceEl.CREATE (-p_k, NewStateNr,
             ActionNr ← 0,
             LexPos ← TheLex.CurrentPosition,
            LexLength ← TheLex.CurrentLength,
            LexCarnation ← TheLex.CurrentIncarnation,
             RHSNr ← -1,
             Λ);
```

Then, we set the system to a stable state, push the newly created reduction, so that the parser can resume.

905c ⟨*Build a phony reduction* 905a⟩+≡
```
TheStateNr ← NewStateNr;
TheActionNr ← 0;
TheStack_i ← ReduceEl;
TheSp ← i+1;
RESULT ← TRUE;
Good ← TRUE;
```

The following chunk is executed if no error-recovery phony reduction could be performed on the current token. We then backtrack lexically, that is, try to see whether there is another interpretation of the current lexeme on the input. If there is no such alternate interpretation, the next token is read from the input.

905d ⟨*Lexical backtrack, or advance* 905d⟩≡
```
TheLex.BackTrack;
IF TheLex.CurrentToken = TheLex.ERROR THEN
  TheLex.ResetMaximumLength;
  TheLex.Advance;
  SkippedTokens ← SkippedTokens + 1;
  ⟨Every so many tokens, dump 906⟩
  END;
```

Just not to let the user with a looping process without any further information, dump something once in a while. It is true however that if skipping more the 2048 tokens did not lead to a successful recovery, chances are that the parser will probably not recover at all.

906 ⟨*Every so many tokens, dump* 906⟩≡

```
IF SkippedTokens MOD 2048 = 0 THEN
  StdOut.WriteInt (SkippedTokens, 0);
  StdOut.WriteString (“ tokens skipped, in position: ”);
  StdOut.WriteInt (TheLex.CurrentPosition, 0);
  StdOut.WriteLn;
  END;
```

7.29.5 Debugging methods

The following chunks are dramatically uninteresting to read.

907 ⟨*Debugging methods* 907⟩≡

```
REDEFINE METHOD DumpRecoverTable(Output: OutputStream);
  VAR
    p: ARRAY OF INTEGER;
    Table: ARRAY OF ARRAY OF INTEGER;
  BEGIN
  Table ← TheRecoverTable;
  FOR i ∈ 0 .. Table.SIZE-1 DO
    p ← Table_i;
    IF (p ≠ Λ) ∧ (p.SIZE > 0) THEN
      Output.WriteString (TheLex.TokenImage(i) + ‘‘:’’);
      FOR x ∈ p DO
            Output.WriteInt (x, 0);
            Output.WriteString (‘‘ ’’);
        END;
      Output.WriteLn;
      END;
    END;

  Table ← TheRecoverSyncTable;
  FOR i ∈ 0 .. Table.SIZE-1 DO
    p ← Table_i;
    IF (p ≠ Λ) ∧ (p.SIZE > 0) THEN
      Output.WriteString (‘‘On state: ’’);
      Output.WriteInt (i, 0);
      Output.WriteString (‘‘:’’);
      FOR x ∈ p DO
        Output.WriteInt (x, 0);
        Output.WriteString (‘‘ ’’);
        END;
      Output.WriteLn;
      END;
    END;
  END DumpRecoverTable;
```

908a ⟨*Debugging methods* 907⟩+≡

```
METHOD DumpLexStatus(Msg: ARRAY OF CHAR);
  BEGIN
  StdOut.WriteString (Msg + “ Image:” + TheLex.Image + “ Code: ” +
                      TheLex.TokenImage (TheLex.CurrentToken));
  StdOut.WriteString (“ Input position: ”);
  StdOut.WriteInt (TheLex.CurrentPosition, 0);
  StdOut.WriteString (“ Token length: ”);
  StdOut.WriteInt (TheLex.CurrentLength, 0);
  StdOut.WriteString (“ Carnation: ”);
  StdOut.WriteInt (TheLex.CurrentIncarnation, 0);
  StdOut.WriteLn;
  END DumpLexStatus;
```

908b ⟨*Debugging methods* 907⟩+≡

```
METHOD DumpStack;
  BEGIN
  FOR i ∈ 0 .. TheSp-1 DO
    StdOut.WriteString (“State: ”);
    StdOut.WriteInt (TheStack_i.FromStateNr, 4);
    StdOut.WriteInt (TheStack_i.ToStateNr, 4);
    WHAT TheStack_i OF
      IN ShiftStackElement:
        StdOut.WriteLine (“ Lexeme: ” +
                            TheLex.TokenImage(TAG.LexCode));
        END;
      IN ReduceStackElement:
        StdOut.WriteString (“ RHS:    ”);
        StdOut.WriteInt (RHSToRule(TAG.RHSNr), 0);
        StdOut.WriteString (“ by ”);
        StdOut.WriteInt (TAG.RHSNr, 0);
        IF TAG.NonTerminal ≠ Λ THEN
          StdOut.WriteString (“ -> ” +
                                   TAG.NonTerminal.ClassName);
         ELSIF IsVoid(TheStack_i) THEN
          StdOut.WriteString (“ -> VOID”);
          END;
        StdOut.WriteLn;
        END;
      END;
    END;
  END DumpStack;
```

909a ⟨*Debugging methods* 907⟩+≡

```
$If $TableDriven
  METHOD RHSToRule (RHSNr: INTEGER): INTEGER;
    BEGIN
    IF RHSNr < 0 THEN
      RESULT ← -1;
     ELSE
      RESULT ← TheRHSToRuleNrTable_RHSNr;
      END;
    END RHSToRule;
$Else
  METHOD RHSToRule (RHSNr: INTEGER): INTEGER;
    BEGIN
    CASE RHSNr OF
      -1:
        RESULT ← -1;
        END;
      $RHSToRule
      END;
    END RHSToRule;
$EndIf
```

909b ⟨*Debugging methods* 907⟩+≡

```
$If $DynamicVerboseLevel
METHOD VerboseLevel: INTEGER;
  BEGIN
  RESULT ← TheVerboseLevel;
  END VerboseLevel;


METHOD Verbose: BOOLEAN;
  BEGIN
  RESULT ← TheVerboseLevel > 0;
  END Verbose;
$EndIf

METHOD SetVerboseLevel(Level: INTEGER);
  BEGIN
$If $DynamicVerboseLevel
  TheVerboseLevel ← Level;
$EndIf
  END SetVerboseLevel;
```

910a ⟨*Debugging methods* 907⟩+≡
```
METHOD Warning (Message: ARRAY OF CHAR);
  BEGIN
  DumpLexStatus (“Warning ” + Message);
  END Warning;
```

7.29.6 Measuring parsers

The following chunks implement the various measuring features of the parsers DURA generates. These metrics will allow us to measure the gap (if any !) between the theoretical exponential performance of backtracking-based systems, and the reality of real-world languages and input.

First, the attributes...

910b ⟨*Attributes and constants* 868b⟩+≡
```
VAR
  TheShiftCount,
  TheUnShiftCount,
  TheLexicalBacktrackingCount: INTEGER;
  TheReduceCountArr,
  TheUnReduceCountArr: ARRAY OF INTEGER;
```

Then, the chunk that resets these attributes when initializing the parser:

910c ⟨*Resetting metrics information* 910c⟩≡
```
TheLastPos ← 0;
TheShiftCount ← 0;
TheUnShiftCount ← 0;
TheLexicalBacktrackingCount ← 0;
TheReduceCountArr.CREATE ($MaxRHS + 1);
TheUnReduceCountArr.CREATE ($MaxRHS + 1);
```

910d ⟨*Update statistic for unreduces* 910d⟩≡
```
TheUnReduceCountArr_j ← TheUnReduceCountArr_j + 1;
```

910e ⟨*Update statistics for unshifts* 910e⟩≡
```
TheUnShiftCount ← TheUnShiftCount + 1;
```

910f ⟨*Update lexical backtracking statistics* 910f⟩≡
```
TheLexicalBacktrackingCount ← TheLexicalBacktrackingCount + 1;
```

`StatAcc` returns the various statistics information of the parser at the current stage, aggregated as an instance of the `ParserStatAcc` class (See page 361), so that they can be accumulated to compute aggregate stats on several parsing runs.

911a ⟨`StatAcc` *method implementation* 911a⟩≡

```
REDEFINE METHOD StatAcc: ParserStatAcc;
  BEGIN
  RESULT.CREATE;
  RESULT.SetReduceCounts (TheReduceCountArr,
                          TheUnReduceCountArr,
                          TheRHSToRuleNrTable);
  RESULT.SetShiftCounts (TheShiftCount, TheUnShiftCount);
  RESULT.SetLexicalBacktrackingCount
                          (TheLexicalBacktrackingCount);
  RESULT.SetLineCount (TheLineCount);
  RESULT.SetErrorCount (TheErrorCount);
  END StatAcc;
```

The following chunks implement an alternate skeleton, that can be used to generate an empty parser, for instance, in early stages of development when we basically need a parser module, for the project to compile, even if no grammar information is available.

911b ⟨*dempty.dura* 911b⟩≡

⟨*dskel.dura* 865⟩

911c ⟨*iempty.dura* 911c⟩≡

```
IMPLEMENTATION MODULE $ParserModule;

CLASS $ParserClass;
  INHERITS BaseParser ($RootModule.$RootClass, $LexerModule);

  REDEFINE METHOD CREATE;
    BEGIN
    END CREATE;

  REDEFINE METHOD Parse(Lexer: $LexerModule): $RootModule.$RootClass;
    BEGIN
    END Parse;

  END $ParserClass;

END $ParserModule;
```

Well, this is the end of the skeleton. It is of course a very complex and lengthy part of this work. This is where most of the tricks we talked about are implemented. Under normal circumstances, we try to make the system robust enough, so that plain users don't have to get a firm grip on this skeleton's internals to use DURA effectively. This is no trivial goal by any standard. However, when parsers have strange behaviour, this annotated source code can prove a very valuable asset to better understand what is actually hapenning.

Chapter 8

Results

This section describes the industrial and academic results obtained using `lexyt` and `DURA`. It is of course a rewarding experience for an academic project such as this one to be used in real-world environments to solve real-world problems. It gives some insight on how relevant some of the original design decisions might have been. It also shows by means of example and production-level testing that the implementation has left the prototyping stage. `DURA` and `lexyt` are still evolving to provide the facilities required to produce parsers for more and more exotic languages, but such evolutions have become rare. `DURA` and `lexyt` are stable, and can be used for projects where one cannot actively support the tool in addition to all other issues that have to be taken into account. The tool is supposed to be a solution. Not part of the problem any longer.

It is true that real world usability has been a goal from the whole beginning, when starting this project. However, I never thought we would have as many opportunities as we had in the last two years to actually use the technology. In the case of `RainCode` (See page 920), it is the mere availability of the technology that was one of the major motivations for the development of the product.

One of the most exciting aspects of this migration from the prototype stage to a production-level tool was the ability to have other people than myself using the tool; to see how they understood the documentation, how they extrapolated their experience with `yacc` to `DURA`, how they bumped into problems I avoided by knowing `DURA`'s internals. Their feedback has been invaluable. The tools had to be made more robust in various areas to better detect and report common mistakes (such as obscure ways of ending into a grammatical cycle).

Several projects are listed here, in chronological order. They range from short-lived industrial projects where parsers had to be produced and thrown away after a while, to the development of a far more ambitious product, where the experience gained in previous projects has been used profitably.

8.1 Fortran

We developed a FORTRAN parser within a project for the CERA [91] bank. The original assignment was to improve the performance of a large risk analysis application that had been developed by skilled domain specialists who did not have any formal training in computer science. Issues such as performance analysis and, more generally, software engineering were as remote to them as mint sauce recipes are to me. In other words, it was anarchist development, grown from a prototype that was never meant to go into production, but which eventually did as a result of management-related pressures.

The most obvious sign of this limited insight for software development showed in the lack of modularisation: the entire system was made of a single FORTRAN module of over 35.000 lines. Several people worked simultaneously on the system and the various versions of the source file were merged together manually. Quire obviously, they had far harsher problems than performance alone. Incidentally, as it is often the case, these developers who did lack formal knowledge about complex software systems had biased ideas regarding their work. For instance, they were intimately convinced there was no significant factor of performance to be gained. They were convinced their code was close to optimal.

Most of the performance issues have been solved within 15 minutes, by showing the developers (well, of course, they were not professional software developers, but since they developed it, referring to them as developers is fair) what a profiler was. Using the profiler bundled with the compiler they were using on a single run of the software showed major bottlenecks that accounted for over 80% of the total execution time, and that could be optimized easily. Within a couple of days, they had achieved a tenfold performance improvement by themselves.

We then concentrated on the quality of the system in order to improve its maintainability. The software had several interesting properties, some of which were due to the little technical expertise of the developers:

- The code was entirely structured. There was not a single GOTO in the code
- As noted above, it was made of a single huge module
- It was made of over 200 functions, some of which were quasi duplicates of others, with a small variation. A more thorough analysis might have lead to merge some of these functions, but due to time constraints, we have not been able to pursue this direction any further
- Most of the large data structures were dynamically allocated arrays of simple elements. It soon appeared that some of these arrays of simple elements were to be understood as an array of structures. Simply, the developers had no knowledge of what a structure was, and developed the system using their level of expertise.

Considering this, we designed a strategy as follows:

- Modularizing was a must. This obvious improvement was required to break up the huge module in smaller, more manageable components.

- This modularization should not be brute force. It is of course possible to define a module for each function, but we believed it would not have helped much in terms of maintainability. We wanted something more flexible, where modules could be defined as the reunion of semantically connected functions.
- Some of the separate arrays had to be grouped into arrays of structures.

The challenge here lied in the fact that we wanted these alterations to be performed automatically by a tool, that had to be developed within a rather short time frame.

We developed a `FORTRAN` parser using `DURA`. It included a primitive form of semantic analysis, that yielded a call graph for functions, and a usage list for variables. Based on configuration files that defined how modules had to be grouped together, and how separate variables had to be grouped into structures, the tool could then change the source code without further manual intervention.

8.1.1 Dealing with modularity

One issue we did not pay enough attention to, was the fact that modularity as supported by `FORTRAN` (at least, in its `FORTRAN90` incarnation) does not allow cyclic imports. Consequently, just any partition of the original module in smaller pieces is not equally valid. Fortunately, the developers did not use recursion. Hence, we had at least the guarantee that such a valid partition was within our reach, since the naive partition where each function is stored into a separate module would have been valid, if not very useful.

We spent quite some time tuning the module structure, submitting the resulting architecture to the `FORTRAN` compiler to see whether it would compile. It has proven to be far more awkward than originally thought. The lesson we have learned from this experience, is that it would have been far easier and more productive to design a tool that used dependency information to assist in this process of the partition of the huge system into manageable entities. Instead, we went though the modularization effort through a cumbersome manual trial and error process, which required far more time than we expected.

In conclusion, the code restructuring tool we developed worked, and produced a modularized version of the system, that has been compiled, and worked flawlessly at the first attempt, except for a single case that could not be handled by the tool due to a lack of orthogonality in `FORTRAN90`.

8.1.2 The patch mechanism

Performing automated transformation on source code at the textual level is simple in theory, and can prove awfully cumbersome in practice. If implemented naively, the entire process must be entirely built around a loop that reads the original source code, and takes appropriate action – such as insertion, deletion, replacement – when various kinds of components are found.

In the course of this project, we had to handle patches far too complex to support this straightforward scheme. Therefore, we built an abstraction layer that hides most of the complexity related to patches and that provides and extra level of flexibility.

This layer maintains patches, that is, insertions, replacements, and deletions, in a waiting list. Patching is then merely a question of adding a patch to this waiting list. These patches can be added to this list in random order, thereby lifting all constraints in terms of the order in which things must happen.

Then, when all patches have been applied, that is, when all patch objects have been added to the patch list, this list is sorted according to the position of the patches, the input file is read, and all the patches are applied in sequence to produce a patched output file.

This abstraction of the patch mechanism simplified the problem considerably. The main reason for this lies in the fact that the various patch operations (insertions or deletions of large chunks of code) do not alter line numbering until the patched output file is being produced. Patches can occur in any sequence, and don't have to take previous patches into account to determine positions.

8.1.3 Producing HTML-based documentation

At the end of the project, we still had some time available, so we extended a bit the scope of the project, and we produced HTML based documentation. An HTML [2] file was produced out of each source file where clicking on an identifier links to the corresponding declaration (variable or function). Besides, for each variable or function, the list of usages could be obtained as well. All in all, this tool proved very valuable when maintaining and reading a large and complex system for which no structural documentation existed otherwise.

As a nice side effect, we also produced tables summarizing the size and complexity of all the functions and modules in the system, to direct the maintenance efforts towards the most entropic pieces. These tables were generated by inserting HTML tags using the patch mechanism described above. Some functions were over 2000 lines long. As it is often the case, such quality-directed information has not been taken into consideration by the developers, arguing that there was no time left. Software engineering sometimes feels like fighting windmills [92].

8.1.4 The lessons we learned

Besides the detail about acceptance of the quality-directed information by the developers, several lessons have been learned in this first application of `DURA` to real-world problem.

- When working in this project, `DURA` did not have the automatic hierarchy handling facilities described in 7.5.3. This was a major shortcoming, and this was the major motivation for the addition of this feature to the parse trees produced by `DURA`. At the time, this feature was emulated using a brute-force method that did the job, but that would have been inappropriate if the tool was not a short-lived utility to throw away at the end of the project.

- The major problem we encountered was plain logistics: at the time, `DURA` did not support interpreted parsers as described in 6.2.1 and the parser with the entire transition logistics builtin as nested CASE statements was painful to compile, to say the least. On state-of-the-art machines of 1997, these compilation times severely hampered productivity, for a runtime performance gain that we could not evaluate at the time, and that was probably not even necessary. A few months later, we implemented interpreted parsers, and they are used systematically now, for virtually all the parsers we generate using DURA.

- FORTRAN is notoriously difficult to parse, because it does not support reserved words. There is nothing that prevents the user from calling a variable IF, with all the painful consequences on the parsing algorithm one can imagine.

  Since the tool we developed was meant to be used within the scope of a single software project, we did not have to take the full generality of FORTRAN90 into account, and restricted ourselves to a sane subset of the language, where, among other similarly simplifying assumptions, keywords were *reserved*, and would not be used by programmers for user-defined purposes.

  Alas, whenever a degree of freedom opens itself, no matter how bizarre it is, there is always a way in which programmers start exploiting it. Software is just like gas: as soon as there is more than just a bit of it, all possible degrees of freedom are used at least once.

  In this specific case, there is a place in the Microsoft standard library where a structure is defined, and where a field of this structure is named TYPE, which is a FORTRAN90 keyword, and which was used to declare the data structures in the application at hand. Hence, when using this structure, one uses a keyword as a user-defined identifier. When we encountered this very specific case, all we had to do was to change the lexical analyzer to support level 1 backtracking instead of level 0 (See page 29), and the parser worked flawlessly. To fully recognize FORTRAN90 according to the reference manual, a lexer with backtracking level 2 would have been necessary.

- The most obvious added value of DURA when compared to other parser generators such as yacc lies in its ability to analyze languages that do not enter one of the commonly accepted classes (LL1, LALR) but even more, in its ability to produce a parse tree automatically instead of requiring mundane and error-prone tree-creation code. This was a critical factor to produce a working parser for FORTRAN90 given the time constraints we had to comply with.

8.2 ObjectZ

Z [93] and its object-oriented extension ObjectZ [75] are specification languages based on first-order logic for which executability has never been perceived as an issue, even if some recent efforts have been invested into type-checkers [94] [95] first, then animators [96], and finally theorem provers [97] for Z.

However, this lack of executability meant that these languages have remained at an abstract level for quite some time, and due to the fact that they included enough uncommon glyphs to make even APL [9] blush, the first *implementation* of Z was made of a bunch of LaTeX macros to assist in the – admittedly, harsh – task of Z typesetting. There was absolutely no Z-related intelligence in these macros. They were simply typesetting macros, with no interpretation nor control on the underlying Z semantics. There was no distinction between writing and typesetting an ObjectZ specification.

This modus vivendi has extended into a de facto standard by now, and virtually all Z-related tools today use the LaTeXmode as input (there are exceptions, though: [95] and [96] and each define their own textual input format).

Together with Jean-Christophe Réal, we investigated the ability to parse ObjectZ specifications, and we ended up checking an implementation of such a parser included in the Wizard [98] type checker[1].

Wizard's parser was quite complex, generated using the LALR/1 parser generator included into the Cocktail toolbox [99], and suffered from some minor limitations. It is based on a very complex lexical specification, where lookahead facilities (similar to what is described in for `FORTRAN`, see figure 2.5 in page 24) are used extensively to distinguish between different usages of the same lexeme, and return different codes to the parser depending on complex contextual information. Even so, the parser included in Wizard failed to parse some of the simplest examples included in the original ObjectZ distribution.

While we have no definite evidence of this, we have the feeling that this complexity (as well as the parser's limitations, by the way) are due to the LALR/1 limitation of the underlying parser technology.

It soon appeared to us that rewriting the parser using `DURA` would be far easier and would provide more added value than trying to reuse the existing Wizard parser. It is a rather unexpected area of application of more sophisticated parser generation technologies. We expected the `DURA`-generated parser to be far more versatile and easier to maintain and to extend than Wizard's[2]. While we cannot speculate about how hard it would have been to use the Wizard parser as starting point, dealing with the `DURA`-generated parser on the longer term has proven to be very comfortable.

One must understand that this is by no mean a simple challenge. The information on the formal grammar of ObjectZ is diluted in various presentation details, some operators are prefix, some others are postfix, their precedences are not even specified in the language definition. None of the commonly applied restrictions (LALR, LL) is applicable[3]. Quite understandably, no textbook (not even ) mentions ways or tools to develop such parsers.

It is also true that this was for me the first experience of giving `DURA` away, that is, having another user than myself or someone working directly with me[4]. This factor, together with the technical nature of the job, made the development of this parser a very interesting experience indeed.

[1]Object-oriented extensions to Z induce differences in the type system that seem to require an entirely different kind of technology than what was used to typecheck Z.

[2]As a matter of fact, the actual redevelopment of this parser has been done by Jean-Chrisptophe Réal only. I merely assisted him on a few technical details and basic design decisions.

[3]As confirmed by[100]'s comments attached to its PRECC [61] [62] based grammar for Z.

[4]At the time of the ObjectZ parser project, Jean-Christophe was not working at out premises, even if we had very frequent, friendly and fruitful contacts.

8.2.1 The parser

The first step of our parser is a preprocessor that removes all the typesetting commands that are not relevant to the semantics of the specification (comments, tab positioning, page or column breaks, etc.), and that replaces them by the same number of white spaces. Thereby, the absolute positions of the relevant parts of the text are not altered. The error-reporting information that will be produced by the further passes of the parser will be accurate in terms of line and column numbers.

At first sight, it might be tempting to just remove the invocations of all LaTeXmacros that are not recognized by the system in a generic way. However, this scheme, while tempting, cannot be implemented due to the way TeX (and, consequently, LaTeX) handles parameters.

When writing something of the form `\johndoe{arg1}{arg2}{arg3}`, the number of arguments attached to a command invocation is only defined in the `johndoe` macro definition, not by the way it is invoked. If the macro was defined as having a single parameter, `\johndoe{arg1}{arg2}{arg3}` is an invocation of the macro `\johndoe` with `arg1` as argument, followed by two loose pieces of text `arg2` and `arg3`. Consequently, we cannot remove the invocation and the parameters without having access to the macro definition. Hence, the preprocessor takes a configuration file with macro names to be removed from the input file, *together with their arity*.

Besides the preprocessor issue, the only trouble Jean-Christophe had when developing the parser was to decrypt the parser included in Wizard, where the same lexeme returned different tokens depending on their lexical context in order to lift syntactical ambiguities. Such practices made the existing grammar far harder to read than it should have been.

It is also true that he suffered from quite some problems because of the relatively poor quality of the documentation for `DURA` and `lexyt`, and their relative lack of robustness at the time.

8.2.2 The lessons

The ObjectZ parser gives poor but predictable performance when dealing with a correct input. Things get badly out of control when an erroneous input is provided, where parsers backtrack for minutes before giving up and issuing a (by the way, cryptic) error message. In order to address this problem, two solutions have been implemented:

- Limiting the number of backtracking steps. The principle is rather simple: when the parser has performed a given number of steps (understand, shift, reduce, unshift or unreduce) without advancing the input pointer, it assumes that it will not be able to find a possible interpretation of the input, and stops with an error message that indicates the *most advanced position as the source of the error* (See page 894).

- Controlling the cut level (snips and cuts, see 6.9.2) that limits the number of steps performed by a backtracking parser when attempting to reinterpret a construct for which no alternate interpretation can be found.

The second lesson was simpler: the tools had to be made more robust, their error reporting had to be improved, the documentation had to be completed by numerous examples for an external person to be able to deal with them. However, this experiment, and even more, the fact that after encountering a few initial problems, Jean-Christophe got through it rather successfully gave us more and more confidence in the tools – understand: `DURA` and `lexyt` – and in their ability to deal with complex and hard to solve problems.

The last issue that has been raised by the ObjectZ experiment related to performance. The problems we encountered in case of failure to parse have been solved by the changes listed above. The performance issue in case of success is more of a concern and is illustrated in table 8.1 on page 944.

8.3 RainCode

We started developing `RainCode` [101] around May 1998, when the `DURA` toolset was aleady mature and had been used succesfully in several occurrences. We had a working `COBOL` parser, which we used mainly for regression testing purposes whenever a significant change was introduced in `DURA` or `lexyt`. The idea of `RainCode` arose when we realized that the possible things one could be willing to do with `COBOL` source code were so different, so environment-specific (cross-reference relative to copy books or databases, source code tranformation, quality assessment, coding guidelines enforcement, migration, inventory), that no single tool would ever cover them all. So, instead of providing a product that actually *does* things on source code, we started developing `RainCode`, which is a tool that provides the user with high-level abstractions that can be used to implement ad hoc, specific source code analysis and transformation facilities.

In practice, `RainCode` reads `COBOL` a source file, builds a full parse tree, and provides a high-level scripting language which provides convenient primitives that deal directly with the parse tree. It is more than a `COBOL` parser toolbox, as provided by CobolTransformer [102], in the sense that one can express things in the `RainCode` scripting language in a small fraction (often, below 10%) of the code required to achieve a similar level of functionality using `C++` – which is CobolTransformer's operating language. Besides, the `COBOL` parse tree which is manipulated by `RainCode` is not limited to a pure syntax tree, it also includes basic semantic information – attaching all identifiers to the variable or label they refer to – which allows the development of simple but powerful scripts. For instance, figure 8.1 depicts a script which lists all unreferenced labels in a `COBOL` program. One can see in this example the influence of YAFL on the `RainCode` scripting language, including quantifiers [83]. The only significant difference lies in the fact that while `RainCode` requires explicit declaration of all the variables, these declarations do not indicate a type (as for `Ids` in figure 8.1), since a variable can hold values of different types over its lifetime, without restriction: integer, character string, list, boolean, non-terminal, etc.

We also integrated the patch mechanism (as described for `FORTRAN90` in 8.1.2) in `RainCode`, and made it accessible from within the scripting language, so that one can use `RainCode`, not only to *find* things in `COBOL` programs, but *change* them as well.

Figure 8.1 A simple RainCode script

```
PROCEDURE DumpUnusedLabels;
  VAR
    Ids;
  BEGIN
  Ids := ROOT.SubNodes | X IS LabelIdent;
         -- "|" implements the filter operation to restrict a
         -- list to those of its elements that fulfill
         -- some condition, using the predefined variable X as
         -- free variable
  FOR Bl IN ROOT.ProcedureDivision.Blocks |
     NOT (THERE_IS Id IN Ids :- Id.Ref = Bl) DO
    OUT.WriteLn ("Unreferenced label: ", Bl.Label,
                 " in line ", Bl.LineNr);
    END;
  END DumpUnusedLabels;
```

8.3.1 Overall architecture

There is more in RainCode than DURA and the patch mechanism only. Before even starting the development, we identified three major problems we would have to deal with:

- The interpreter for the scripting language must have decent performance. Some of the needs it is meant to address might require thousands of operations, that must be executed within reasonable timing constraints. Failure to achieve decent performance would invalidate the RainCode concept entirely, by making it unpractical when compared to a parser class library to be interfaced from within a third generation language.

- When developing an interpreter, one does not only provide a *language*, but also, a possibly large number of primitives, predefined functions (or verbs, classes, operators, depending on the scripting language at hand). One must provide some *glue* between the compiled world – where the primitive is implemented – and the interpreted world – where the primitive can be used – to give the user access to these predefined functions. This glue presents no difficulty except for size. It is simple to write, but managing these hundreds of bridges between the two worlds, compiled and interpreted, specially in a language such as YAFL which does not provide anything like pointers to functions *a la* C, can become a real mess.

- It is not enough to provide these hundreds of primitives, one must also document them, and working on the implementation for weeks, then starting a word processor to write 500 pages of documentation is a fantasy. It simply does not work this way. We had to imagine ways – other than hiring a dozen technical writers, of course – of producing good quality documentation within acceptable delay constraints.

The careful reader might have noticed that all these problems we identified as being essential to the development of `RainCode` bear some relationship with the interpreter. The reason for this is very simple: `RainCode` is little more than an interpreter for a specialized programming language which operates on which basic data structures that happen to be parse trees. Since the delivery of the parse trees is taken care of by `DURA`, the remaining issues deal with the interpreter.

8.3.2 Building an efficient interpreter

The literature commonly makes a difference between compilers – that translate to something, usually native code for a target architecture – and interpreters – that execute something. This dichotomy usually comes with a commonplace implication about interpreters being slower than compiled code.

This simplistic classification is a bit rough on the edges. There are many different forms of interpretation, with very different characteristics and behaviours:

- Some interpreters (such as Perl [103]) generate an intermediate representation on the fly. Syntax and even a bit of semantics are checked for before starting to execute. Execution performance is thereby improved. Statements are not analyzed in tight loops over and over again before each execution. When generating p-code, this approach is vastly similar to plain compilation, except for the fact that code generation usually is much simpler, and the performance penalty due to the interpretation of the p-code. Despite the claim that Java [104] is compiled, it actually lies more or less in this category. A virtual machine is just another word for p-code. It is by the way very typical that most of the effort toward improving Java's performance goes in optimizing the virtual machines rather than the compilers.

- Some interpreters take a radically opposite approach. Input is considered line by line, and full analysis (lexical, syntactical, semantic) is performed again every time. This approach proves efficient when dealing with a large program out of which only a small subset of the statements gets executed on a typical run. This is how a language such as Python [11] [12] works. Such interpreters generally allow the dynamic construction and execution of code fragments during the execution of a program.

- When a program makes use of very high level abstractions provided by the language, the cost of decoding the instruction can be negligible compared to the cost of executing these high-level primitives. This is typically the case of matrix operations in APL [9], where tight loops are executed efficiently within the interpreter's primitive rather than in APL source code.

`RainCode` uses a variation of the first scheme, in the sense that full syntactical analysis is performed when loading the script, but execution is performed directly *on the parse tree* produced out of the script. It actually executes nodes in a parse tree produced by `DURA`. This execution is not performed by a big loop that performs a huge

WHAT[5] on each node, and takes appropriate action depending on the class of the node, but rather, directly in the nodes themselves.

For instance, all non-terminals which inherit from the `Statement` base class must redefine an `Execute` method as in figure 8.2, all non-terminals which inherit from the `Expression` base class must redefine an `Evaluate` method, etc.

The `WHAT` statement's performance is linearly correlated to the number of classes tested, while the cost of late binding is constant, independantly from the number of polymorphic classes.

Figure 8.2 The source of the `RainCode` `IF` statement

```
NONTERMINAL CLASS IfStatement;
  INHERITS Statement;

  GRAMMAR
    If TheCondition Then
      MainBody
      (Else ElseBody)?
      End;

  VAR
    MainBody, ElseBody: StatementList;
    TheCondition: Expression;

  REDEFINE METHOD Execute (Context: RcContext);
    BEGIN
    IF Context.EvaluateBoolean (TheCondition) THEN
      MainBody.Execute (Context);
     ELSIF ElseBody <> VOID THEN
      ElseBody.Execute (Context);
      END;
    END Execute;

END IfStatement;
```

Such a construction requires a bit more memory than the other approaches, where one can concentrate on the definition of a compact and efficient instruction set for the virtual machine in order to reduce the storage requirements of the intermediate representation. In practice, in `RainCode`'s context, this seldom is a limitation given the size of a typical `RainCode` script (Under the 1000 lines).

On the other hand, it offers major advantages:

- It does not require the definition and implementation of an abstract machine nor the code generation to translate the parse tree into this abstract machine
- Interpretation happens at the highest level of abstraction possible

[5] The `WHAT` construct tests the dynamic of an instance in YAFL. It is somehow similar to RTTI as supported by `C++` [105] or to the `instanceof` operator as provided by Java [104].

- The performance is good, since the *selection* mechanism that decides what to do depending on the statement to execute is implemented using late binding, which is very efficient in YAFL. For instance, considering figure 8.2, the selection of the proper version to execute when applying `Execute` on a `Statement` is a highly optimized YAFL primitive.

It should finally be noted that in the glorious world of compilers and interpreters, performance is not only affected by high-level design decisions. Small implementation tricks can have more influence on real performance than vast, system-wide plans.

8.3.2.1 Implementing a value-based semantics

`RainCode`'s scripting language has a value-based semantics. Parameter passing always happens by value. When dealing with the only composite type it handles – namely, lists – the same rule prevails: no statement can *change* a list. One can just compute a new list based on the elements of an existing list, adding or removing elements.

One might be tempted to implement this value-based semantics, using an equally value-based *implementation* of the interpreter. Experience seems to show that this is not the way to go.

Passing an integer by value to a YAFL method (hence, to a `C` function) is very efficient. On the other hand, `RainCode`'s scripting language is interpreted, and supports dynamic typing. As a program is being executed, a variable can hold values of different types. To implement this feature, all values – including plain integers – are implemented by a number of classes which inherit from a common base class. Copying an instance of a class is significantly more costly than copying an integer, since it requires a dynamic allocation, and besides the intrinsic cost of allocation per se, it also increases the frequency of the garbage collector's mark and sweep process. Actually copying an object everytime an integer is passed as parameter would be a very poor design decision.

To overcome this limitation, the values as handled by the `RainCode` interpreter are immutable: they cannot be changed after creation. Such a property is easy to ensure using strong encapsulation as provided by YAFL.

Since the values cannot be changed, it is now safe to pass them by reference *while ensuring a value semantics*. When performing an assignment of the form `a := b`, there is no need to create a new value for `a`: `a` and `b` can share a reference to a common, immutable object. If `b`'s value must change, for instance, when executing `b := b + 1`, the original value `b` and `a` refer to is not altered: `b` as a reference is altered to refer an instance with the proper value.

Of course, one might object that by not reusing the objects that represent values when requiring new and different values, we put a significant strain on memory management. In practice, having immutable values requires far less allocations than mutable values that would be passed effectively by value – understand, by performing an explicit copy. Besides, `RainCode` has a few other ways of reducing the number of allocations even further.

At initialization time, `RainCode` pre-allocates the values for `TRUE`, `FALSE` and all integers ranging from -512 to 512. Whenever an integer value within this range is needed (and this will be the case in the vast majority of the cases that occur in practice), a reference to one of these pre-allocated values will be used, thereby significantly reducing the number of allocations performed on the fly by the system. Similarly, whenever a constant string value is used in a program (for instance in `b := "Hello world"`), the corresponding value's object is preallocated so that a loop such as

```
FOR i := 1 TO 10000 DO
  b := "Hello world";
  END;
```

does not allocate 10000 identical values. This should be compared with YAFL, where, unlike `RainCode`, instances are mutable. Hence, a new instance must be allocated whenever a string literal is encountered, and the loop depicted above would produce 10000 separate allocations.

8.3.2.2 String spaces and hash keys

The `RainCode` scripting language is interpreted. When referring to a variable at runtime, the system will look for a matching declaration. Our experience seems to indicate that this is probably the most performance consuming process in an interpreter, and this is where effort can be invested fruitfully to improve performance significantly.

The cost of naively implemented symbol tables using linear data structures[6] would be totally unacceptable. For instance, hash tables can be used to deliver better performance. For those of the readers who went through the source code included in this work with a reasonable level of detail, it is clear now that hashing is a personal favorite, and even more so in this context for the reasons listed below.

When using hashing to find identifiers in a hash table, the performance of the *hash function* is a major issue. It is a matter of a good tradeoff between the best possible hash function, and the induced performance penalty when using a complex, well-distributed function compared with a more naive scheme with less attractive properties.

In order to lift this issue entirely, the hash values attached to identifiers are computed on the fly when parsing the script, and stored together with the idenfitiers in the parse tree. The hash function is never invoked after that, since the values stored in the tree can be used instead. This enables us to use a hash function of arbitrary complexity, since the number of times it will be used is a function of the script size – much smaller than the total number of identifier lookups in a typical script execution.

In addition to the precomputation of hash keys, the `RainCode` interpreter also uses string spaces (as described for YAFL in [31]) to store the character strings of the

[6] It is perhaps worth mentioning that the early YAFL compilers used such linear data structures for symbol tables for the sake of simplicity, based on the assumption that given the very nature of the language, the symbol table attached to every block level would not be very large. As soon as we had a working profiler, it appeared that searches in the symbol table could account for over 50% of the total execution time. As confirmed by real-world examples, this figure is far worse when dealing with an interpreter, since the number of times the symbol table is accessed is a linear function of the number of executed statements rather than the size of the program at hand.

identifiers of a program in such a way that reference-based equality can be used in place of costly string comparison. Hence, looking for an identifier in a hash table will be a matter of checking whether the entry corresponding to the precomputed hash value is used, and if it is, ensuring that this entry matches the identifier we're looking for by a reference comparison. The hash tables are sized in such a way that collision hardly occurs.

8.3.3 Predefined primitives

Providing predefined primitives (procedure, classes, functions, variables, depending on the nature of the interpreted language at hand) is not a complex issue. It is merely a question of size and convenience. If we only have to deal with a dozen of such primitives, any brute force scheme will do. On the other hand, `RainCode` describes a `COBOL` program using hundreds of classes, and thousands of primitives, which clearly require a more structured and scalable approach.

When using a rather low-level language such as `C`, a popular approaches is to maintain a table where the name of a predefined function in the interpreter language can lead to a pointer to a `C` function that actually implements the primitive.

Figure 8.3 The table of predefined primitibes

```
typedef struct
  {
    char *name;
    void (*func)();
  } PRIMITIVE_ENTRY;

static PRIMITIVE_ENTRY primitives[] =
  {
    {"SIN", calc_sin},
    {"COS", calc_cos},
    {"PRINT", do_print},
    {"LIST", do_list},
...
    {NULL, NULL}
  };
```

assuming the `calc_sin`, `calc_cos`, etc. are `C` functions.

Adding a new primitive is just a matter of adding an entry to this table. However, this is not the end of the story. It works fine with functions, but it cannot handle classes – assuming of course that the interpreted language we're dealing with manipulates classes. Besides, passing parameters is not entirely benign either. We have to deal with the impedance mismatch between our compiled world – well-defined types, static arity – and the interpreted world – dynamic and polymorphic types, functions can take a variable number of parameters, functions must be able to provide a return value of different type depending on the function at hand or its actual arguments, etc.

RainCode is written in YAFL, and must comply with its limitations. For instance, YAFL does not have such a thing as pointers to functions. Of course, one can emulate them by defining classes which inherit from a base class and which redefine a given method. In theory, it would work fine. In practice, having to define a new class for every primitive we wish to supply in our interpreter would be an intolerable burden.

Therefore, RainCode uses an entirely different scheme. To *publish* primitives, that is, make compiled YAFL entities available from within the interpreted RainCode world, it takes advantage of the fact that the classes it must provide to the user – that is, DURA classes which describe elements of parse trees – are originally YAFL classes. It uses the metaclass facility as provided by the YAFL runtime system [31] (Similar in principles to the reflexive API provided by Java [104]) to *publish* the classes that can be used within the interpreter as all the classes deriving from a given base class. Thereby, by simply providing such a base class, the set of all the classes that can be used by RainCode scripts is determined automatically. It is extracted from the set of all classes defined within the underlying YAFL system.

This idea has been extended, in the sense that the very same metaclass system is used to identify some of these classes attributes and make them available as *methods* that return the attribute's value. These attributes cannot be changed from within the RainCode script, so making them available as methods is acceptable.

Attributes are selected for publication based on a naming convention. If the attribute name begins with the prefix "The", it is made available as a RainCode primitive *without* the prefix.

Figure 8.4 Publishing `DURA` attributes and methods

Publishing an attribute:

```
NONTERMINAL CLASS IfStatement;
  INHERITS BaseNonTerminal;

  GRAMMAR
    If TheExpr Then TheStmt (Else TheElseStmt)?;

  VAR
    TheExpr: Expression;
    TheStmt,
    TheElseStmt: Statement;

END IfStatement;
```

Publishing a method:

```
NONTERMINAL CLASS Ident;
  INHERITS BaseNonTerminal;

  GRAMMAR
    TheImage := Identifier;

  VAR
    TheImage: ARRAY OF CHAR;

  PRAGMA
    Publish(PubNormalized);

  METHOD PubNormalized: ARRAY OF CHAR;
    BEGIN
    RESULT := String.UpperCase (TheImage);
    END PubNormalized;

END Ident;
```

For instance, in figure 8.4, the YAFL (or rather, `DURA`) attributes `TheExpr`, `TheStmt` and `TheElseStmt` are made available from within `RainCode` scripts for all `IfStatement` (or derived) instances as `Expr`, `Stmt` and `ElseStmt`. The name mapping mechanism which, given a `DURA` attribute named `TheExpr`, provides a `RainCode` method named `Expr` is entirely automatic.

Similarly, YAFL methods can be published, by using an ad hoc pragma, as depicted in figure 8.4 as well. As an interesting side effect, the method publishing mechanism is based on the reflexive data structures defined by the YAFL debugger [106] to monitor the runtime behaviour of a running YAFL system. The `Publish` pragma forces the generation of an entry for the method in the metaclass system, but it is not the only way to publish methods: some methods (mainly, public ones) are published automatically by the YAFL runtime system.

In order to better control which methods are made available to `RainCode`, beside mere publication as described above, a naming convention (based on the "`Pub`" prefix) is used here to indicate that a method should be made available from within `RainCode`.

This scheme handles parameterless YAFL methods only. Parameters provided at the `RainCode` level, if any, can be fetched using an ad hoc method (`Args`) by the YAFL method which can then perform various checks – arity, type, etc. The example depicted in example 8.5 shows how the parameter must be fetched by ad hoc code. Since `Args` returns a generic array of value, the `AsString` method must be used to constrain the value onto a character string. It raises an exception if the value it is applied to cannot be converted onto a string in a way which is consistent with the definition of the scripting language. In other words, ensuring that the parameter provided to the `RainCode` method is a character string – or can derive to a character string – is a runtime operation.

Figure 8.5 Reading parameters in `RainCode` from within a published `DURA` method

```
METHOD PubFindBlock: StatementBlock;
  VAR
    a: ARRAY OF CHAR;
  BEGIN
  AssertArity(1); -- Ensures a parameter is provided, raise
                  -- an exception if more or less than a
                  -- single argument is provided
  a := Args[0].AsString; -- Fetch the argument, convert it to a
                         -- character string, raise an exception
                         -- if this conversion is not supported
  RESULT := BlockList.Find(a); -- We're all set, look for the block
                               -- in BlockList
  END PubFindBlock;
```

Return types are mapped automatically from YAFL to `RainCode`: if the YAFL method returns a YAFL character string, a `RainCode` character string will be built automatically, if it returns an integer, a `RainCode` integer will be returned, etc.

All in all, this highly automated interface between the compiled and the interpreted world makes the addition of a new class or method rather straightforward: it is merely a question of adding the corresponding class or method (possibly in the form of an attribute) in the YAFL source code and recompile. It could hardly be simpler.

This approach based on metaclasses is not entirely original. JPython [107] [108] uses a similar system to access Java primitives from within Python scripts.

8.3.4 The documentation system

Producing usable documentation is a challenge by itself. If measured by the number of great softwares for which the documentation is useless, we might even assume that developing working software is the easy part. The real challenge behind software development lies in the documentation.

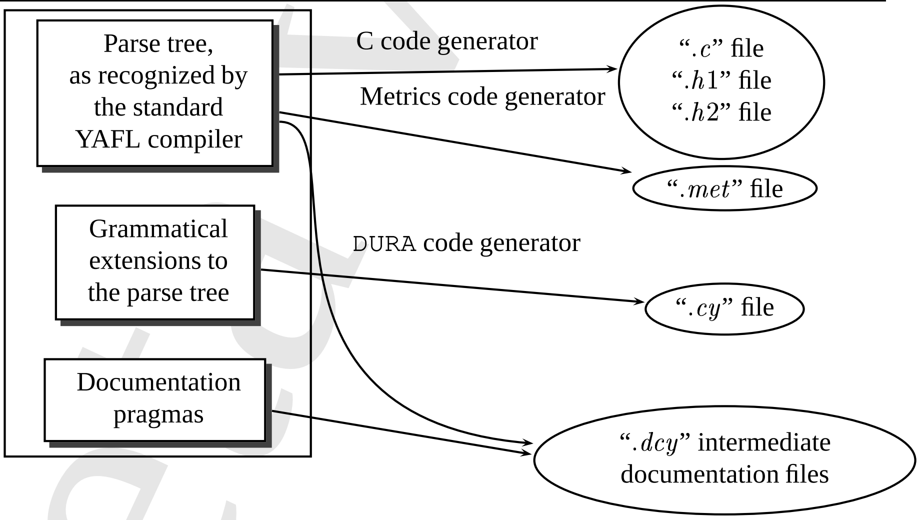
The documentation for a product such as RainCode is made of three parts:

- How to install it
- How to use the scripting language
- The hundreds or thousands of classes and primitives that can be put to work to develop useful RainCode scripts.

In practice, the first two parts together account for less than 40 pages. The only thing that differs from RainCode for COBOL when compared to RainCode for PL/1 is the third part, the class library used to describe resp. COBOL and PL/1 parse trees.

The first two parts can be written in the usual, old fashioned way. They require attention, but are straightforward to write. They don't evolve much over time, and the number of readers per page is high, making the writing experience rewarding[7].

Figure 8.6 DURARC extends DURA (See figure 6.1 on page 257) and generates "$.dcy$" files



The real challenge is to produce workable documentation for the rest, for the class library. Fortunately, most of the relevant RainCode information is basically the immediate translation of equivalent YAFL structures. Hence, we have built an extension of DURA using the compiler support facility as provided by YAFL [31] – thereby producing a three layers compiler. This new compiler, named DURARC (DURA for RainCode)

[7]As an example of the exact opposite of this case, the average number of readers per page for this work will probably lie somewhere between 3 and 4. In other words, an average page that requires one hour to write might just account for up to a cumulated total of a few minutes of – perhaps not so enjoyable – reading.

works just as DURA: it compiles, generates grammatical information, but it also generates documentation information mapped on the structure of class, method and attributes as depicted in figure 8.6. It also compiles user-specified documentation as depicted in figure 8.7, into a consistent whole, saved in an intermediate format, from which printed (through LaTeX [109]) and online (in HTML [2]) versions are then generated automatically.

Figure 8.7 Documentation pragmas in DURARC

```
NONTERMINAL CLASS CallStatement;
  INHERITS Statement;

  GRAMMAR
    CallKeyword
      TheProcToken := (StringLiteral OR DataIdent)
      (WithKeyword
         TheLinkage := StdCallKeyword LinkageKeyword)?
      (UsingKeyword TheArgs := CallArgument+)?
      (GivingKeyword TheGiving)?
      (OnKeyword OverflowKeyword TheOverflowStatement)?
      (OnKeyword ExceptionKeyword TheOnExceptionStatement)?
      TheEnd := EndCallKeyword? ;

  PRAGMA
    Doc(CallStatement);
    DocText("%c denotes a COBOL CALL statement, as in:");
    DocVerbatim('CALL "UPD_CUST" USING CUST-KEY',
                '  GIVING ERR-CODE',
                'END-CALL');

    Doc(TheArgs);
    DocText("%c.%f returns the possibly empty list of",
            "arguments that are provided to the %c.",
            "These arguments are denoted by the",
            ".C CallArgument", "class, that include",
            "the value of the parameter as well as its",
            "mode (reference, value, etc...).");

    Doc(TheProcToken);
    DocText("%c.%f returns a ", ".C StringLiteral",
            "or a", ".C DataIdent",
            "that denotes the routine called by the %c.,",
            "depending on the way this routine name was",
            "specified in the source code.");

    Doc(TheGiving);
    DocText("%c.%f returns the designator in which the",
            "result of the %c must be stored, as specified",
            "by a GIVING clause as in:");
    DocVerbatim('CALL "UPD_CUST" USING CUST-KEY',
                '  GIVING ERR-CODE',
                'END-CALL');
    DocText("If no such clause is present in the source",
            "code, %c.%f returns VOID");
...
```

Figure 8.8 and 8.9 show resp. how printed and online documentation can then be generated from the intermediate representation produced by DURARC.

Figure 8.8 Printed documentation produced by `DURAF`

6.21 CallStatement

Base classes:

BaseNonTerminal (125) Statement (512)

CallStatement denotes a COBOL CALL statement, as in:

```
CALL "UPD_CUST" USING CUST-KEY
  GIVING ERR-CODE
END-CALL
```

6.21.1 Args

Return type: LIST OF *CallArgument (137)*

CallStatement.Args returns the possibly empty list of arguments that are provided to the *CallStatement*. These arguments are denoted by the *CallArgument (137)* class, that include the value of the parameter as well as its mode (reference, value, etc...).

On the example:

`CALL "FORMS$SEND" USING` *BY DESCRIPTOR SESSION_ID*`.`

CallStatement.Args returns the emphasized list made of a single *CallArgument (137)*

On the example:

`CALL "LIB$SIGNAL" USING` *BY VALUE FORMS_STATUS*`.`

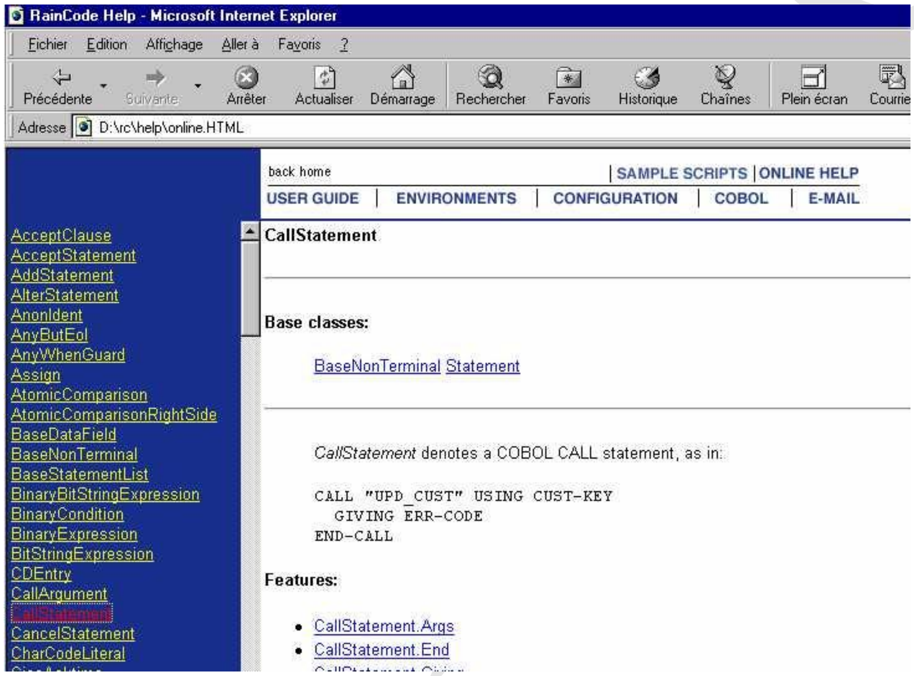
CallStatement.Args returns the emphasized list made of a single *CallArgument (137)*

On the example:

`CALL "TEXTIO"`
    `USING` *TEXTIO-CONTROL*
          TEXTIO-DATA
.

CallStatement.Args returns the emphasized list of *CallArgument (137)*

Figure 8.9 Online documentation produced by DURARC



8.3.5 Documentation by example: one step beyond

At first sight, the documentation generation system as described above is a very efficient way of producing quality documentation with minimal work, since everything that can be deduced from structural information in the source code is managed automatically: inheritance relationships, type of attributes, inherited attributes, etc. The user must only supply pieces of information that cannot be guessed by any automated tool.

However, after having used this system for a few months, I imagined a way it could be extended significantly, to produce even better documentation even faster than we already did. This extension is based on the fact that a large fraction – over 75% – of the methods to document are just attributes which, for a given non-terminal N, return:

- A non-terminal which is actually a subtree of N
- A boolean which indicates some condition about N, for instance, whether an optional keyword has been specified explicitely or not
- An integer which refers to the code of a token of N, for instance to indicate which of two keywords has been specified
- A character string that returns the image of a subtree of N.

Such methods basically provide access to the parse tree in some form. They don't supply intelligence. Most of them are not implemented by YAFL code – understand: YAFL methods – but by access to published attributes which are fetched automatically by `DURA`-generated parsers and published as shown in figure 8.4 in page 928.

These methods are query methods and documenting them is tedious and labour intensive. It is a very systematic and boring process. To automate the documentation of such methods even further, I designed and implemented *documentation by example*.

Documentation by example is a technique where features (procedures, functions, etc.) are documented by providing a number of inputs, together with the result of the application of the feature to the input. It is based on the assumption that by providing these examples and the way the feature reacts to them, one can understand its behaviour reasonably well.

In practice, as implemented within `DURARC`, documentation by example is made of several components:

- I extended the scripting language to include access to the data dictionary. A script can now ask for the list of non-terminal classes in the system, and for each of these classes, the corresponding base class as well as the list of methods that can be applied to instances of this class

- Methods can be called generically. This means that one can write something of the kind of `SYS.Call (Obj, MethName)` to call the method which name matches the content of variable `MethName` and apply it to the non-terminal denoted by `Obj`[8].

- For every language, we maintain a *documentation source*, which is a valid source file for the language, and which contains coded comments containing the "`((-`" and "`-))`" character strings to mark a number of non-terminals (See figure 8.10 on the facing page). Common variations of each kind of non-terminal are typically included in this documentation source, to produce the most versatile and complete documentation possible.

- A language-neutral[9] `RainCode` script, which reads the documentation file, detects the coded comments and builds a list of example instances for each non-terminal class that occur between the starting comment with "`((- BEGIN`" and the matching ending comment with "`((- END -))`".

  For instance, in the example depicted in figure 8.10 on the next page, the script will find all instances of the `ValueClause` class located between the comments "`((- BEGIN ValueClause -))`" and the matching "((- END -))!".

  The script then asks the data dictionary for the list of all non-inherited features for all non-terminal classes for which at least a single example instance has been found. One by one, each of these features is applied to all example instances,

[8] Assuming of course that such a method exists for `Obj`. Otherwise, the script aborts with a runtime error message.

[9] In the sense that it does not include any language intelligence of its own. It is the same script which is used for `COBOL`, `PL/1`, etc. Of course, this script must run under the version of `RainCode` for the language to document, but the script itself is neutral.

Figure 8.10 Source for documentation by example

```
 ...  DATA DIVISION.

 WORKING-STORAGE SECTION.

** ((- BEGIN BaseDataField ValueClause PicClause -))
 01  DICTIO-CONSTANTS.
     02  KEY-SELECTORS.
         03  PRIMARY-KEY         PIC X(1)   VALUE SPACE.
         03  NAME-ONLY           PIC X(1)   VALUE "C".
** ((- END -))
 ...
** ((- BEGIN Statement -))
     MOVE SESSION-TASK-ID
       TO DICTIO-TASK-ID
          LINEIO-TASK-ID
          DIRPK-TASK-ID

     MOVE PROGRAM-LINKAGE TO PARAMETER-LIST
     MOVE ALL PROGRAM-LINKAGE TO PARAMETER-LIST
     MOVE CORR PROGRAM-LINKAGE TO PARAMETER-LIST
     MOVE CORRESPONDING PROGRAM-LINKAGE TO PARAMETER-LIST.
     ALTER CALL-TEXTIO TO ERR-HANDLER.
     ALTER CALL-TEXTIO TO PROCEED TO ERR-HANDLER.
     ASSIGN A-91 TO FILE Z.
     GO TO LAB-VAL1.
     GO TO LAB-VAL1
           LAB-VAL2
           LAB-VAL3
        DEPENDING ON VAL.
     CALL "TEXTIO"
         USING TEXTIO-CONTROL
               TEXTIO-DATA
** ((- END -))
** ((- BEGIN AcceptClause -))
     ACCEPT NO-IMP LINE 22 COLUMN 75
                  WITH CONVERSION DEFAULT IS "1"
     ACCEPT NO-IMP FROM DATE
     ACCEPT NO-IMP FROM VAR-01
** ((- END -))
```

and the results are stored in the "$.dcy$" intermediate format for documentation as depicted in figure 8.6. It is then read by the documentation generation utility and integrated in the manually added documentation. Figure 8.8 on page 932 shows an example of the result obtained by using documentation by example.

8.4 RainCode for other languages

Originally, RainCode was aimed at COBOL only. However, it soon appeared that by careful modularization, one could easily build a framework where by just plugging a different parser, one can get a version of RainCode for a new language in a matter of days. This sections lists some experiences with the development of RainCode for other languages than COBOL.

8.4.1 PL/1

For lexical reasons explained extensively in section 2 starting at page 13, coupled with a high level of ambiguity at the syntactical level, PL/1 is, and quite rightly so, considered as very hard to parse. Since parsing PL/1 was one of the motivations for this entire work, it does not come as a surprise that lexyt and DURA can be used to produce a working PL/1 parser.

One of the basic design decisions underlying RainCode for COBOL lies in the fact that we wanted to have a single product for all available dialects, with all available extensions: SQL, CICS, etc. Basically, RainCode for COBOL recognizes the union of all COBOL dialects we have encountered so far.

Quite logically, we wanted to achieve something similar for PL/1, and even assumed it would be significantly simpler since there are far fewer implementations – hence, dialects – of PL/1 than implementations of COBOL.

Unfortunately, we got stuck on a major dialect difference that requires an *a priori* choice between two parsing schemes to achieve reasonable performance. In other words, the user will have to specify which dialect he (or she) is using be means of a command-line option before actually parsing the source files.

Figure 8.11 show the difference between the two forms of PL/1 IF statements as supported by different dialects.

Figure 8.11 The various forms of the PL/1 IF statement

First form:

```
IF LINE.POS(12) = '0' THEN
  DO;
  LINE.POS(12) = ' ';
  IF LINE.POS(13) = '0' THEN
    DO;
    LINE.POS(13) = ' ';
    IF LINE.POS(14) = '0' THEN
      LINE.POS(14) = ' ';
    END;
  END;
```

Second form:

```
IF CDVRW = 'Q' & DIBSTAT = 'GE' THEN
  POLND = 1;
 ELSE
  POLND = 0;
  IF DIBSTAT = ' ' THEN
    PUT SKIP LIST('ZZZ 'ZCODE' NOT FOUND');
    SWNF = 1;
    RETURN;
    ENDIF;
  ENDIF;
```

A knowledgeable reader would instantly recognize the Pascal [51] or C [110] philosophy in the first of the forms, where one statement is executed if the condition of the IF statement evaluates to TRUE, leaving the possibility of defining *compound statements* to describe behaviours that cannot be described within a single atomic statement.

The second form is faithful to the Ada [111] and Modula II [52] philosophy, where the body to execute if the condition of the IF statement evaluates to TRUE is a *list of* statements, that must be closed by an ad hoc keyword – in this case, ENDIF.

How these two obviously incompatible forms have found place in different dialects of a language while it is clear that they are basically equivalent in terms of effective expressive power escapes me. How could anyone decide to change the syntax of a construct while obviously adding nothing significant in terms of readability or functionality remains a major mystery.

These two forms are almost incompatible – by almost, I understand that a parser that should be able to recognize both of these forms without further a priori information would have an exponential behaviour in some cases.

Intuitively, this can be explained by the need for any such parser for a selection rule, a way of choosing which of the two forms it will consider first, leaving the second form as plan B in case the first, preferred form does not lead to any successful recognition of the input.

For instance, let's consider a partial input, such as:

```
IF A1 = 1 THEN
  B1 = 100;
  IF A2 = 2 THEN
    B2 = 100;
    IF A3 = 3 THEN
      B3 = 100;
      IF A4 = 4 THEN
        B4 = 400;
...
```

These statements are indented in a such a way that a human reader might be lead to believe that they are nested, but a PL/1 parser ignores white spaces except as separators.

The problem arises if the source file and the parser do not agree on which of the two forms depicted in figure 8.11 ought to be tested first. For instance, if the parser first considers the first form – the "Pascal" form – it will consider these IF statements as being performed in sequence, rather than as being nested. Then, when encountering the first ENDIF keyword, it will backtrack and undo the last IF statement to parse it again using the second form – the "Ada" form. When encountering the second of these ENDIF keywords, it will backtrack as far as the previous IF statement, *will try to reparse the last* IF *statement using the first form since this is its favourite form*, backtrack on it again, and so on, going back an forth until all the IF statements of the first form will have been undone, and parsed again using the second form, etc.

More formally, one can attach a bit to each of the IF statements indicating whether the first (bit 0) or the second (bit 1) form has been used, and consider the resulting string of bits for a number of IF statements. For instance, let's consider a case with 8 potentially[10] nested `IF` statements, yielding strings of 8 bits, with a parser that will prefer the first form over the second. When encountering the first `ENDIF` for the first time, we start with a string of 8 0's: 00000000. The last `IF` statement is backtracked onto the second form, yielding 00000001, the second `ENDIF` is encountered, backtracking goes a step beyond, but reparses the last `IF` statement using its preferred form, namely, the first, yielding successively 00000010, then 00000011, 00000100, etc.

The number of steps is thus an exponential function of the number of potentially nested `IF` statements. The other case, where the second – "Ada" – form is preferred while the input is written according to the first – "Pascal" – form is different in nature but similar in process: when encountering the end of the procedure without having found a matching `ENDIF` for all the pending `IF` statements, it will backtrack and reparse for the last of these `IF` statements using the "Pascal" form, will start again until it will encounter the end of the procedure again, fail, backtrack, etc. The resulting behaviour is also exponential.

Since a single `DURA`-based parser cannot parse the two forms transparently while achieving acceptable performance, we are currently investigating the possibility of selecting the preferred form dynamically – understand: after that the parser has been generated, not necessarily after starting parsing – based on a user-defined configuration variable or a command line option. Changing the behaviour of a lexer dynamically is rather simple in many cases, but it is an entirely different story when it comes to something similar with parsers. There are several solutions to this problem, none of which seems entirely satisfactory to us at this stage.

This issue would be far easier to solve using a top-down parser generator such as PRECC [61] [62] (See page 958) where user-defined conditions can be tested *before* starting to consider a grammar rule, while the current version of `DURA` only supports this ability at rule completion.

8.4.2 C, C++ and Java

Probably because `yacc` is based on `C` , or rather, generates parsers written in `C`, `C` is one of the most popular languages to parse in compiler design courses.

This is very unfortunate, because `C` is ill-suited to parsing. It does not match the classical model depicted in figure 2.2 (page 21). One cannot take an independent lexer and parser, and make them work together to produce a working `C` parsing framework without sending some information back from the parser to the lexer.

When writing parsers for `C` (as well as for `C++` [105] and, to a lesser extent, Java [104]) using tools such as `lex` and `yacc`, one must add glue code to handle type identifiers. The grammars for these languages distinguish plain identifiers – for variables and functions – from *type identifiers* – added to the set of usable types using the `typedef` construct. For this scheme to work, type identifiers must be added to an ad hoc symbol table when committing a type declaration. This symbol table must then be accessed by the lexer to return a specific code to distinguish plain identifier from

[10] They are *potentially* nested in the sense that depending on whether `ENDIF` keywords are present, the `IF` statements are nested or not.

type identifiers.

The problem gets worse with the introduction of C++ since one must deal with local classes, hence local types, and deciding whether an identifier is a type identifier or a plain identifier depends on scope resolution, in the numerous forms it can take in C++.

Things would get even worse if one thinks of using a parsing technique that is based on building different parse trees simultaneously in case of a locally or globally ambiguous grammar, as in Earley [30] and Tomita [66]. In such cases, one would need to build and maintain separate type symbol tables for every single simultaneous parse in action. Failing to do so might cause an identifier to be considered as a type, even while building a parse tree for which it would *not* be considered as a type.

The authors of BTYACC [60] claim that given enough lookahead – as in the case of DURA, with infinite lookahead – one can forget this type vs. plain identifier altogether, and solve the issue at semantic analysis time. As shown in [112] this appears not to be as simple as originally claimed. This assertion is based on a simple example, expected in a statement context:

```
p * q;
```

If p is a type, this statement is a declaration of a variable named q which type is a pointer to p. Otherwise, it is a (possibly user-defined) multiplication of a variable p by another variable q, the result of this operation being dicarded.

A BTYACC-generated parser can of course parse this construct in all cases, but without additional information about the exact nature of p, there is no way it can create the right kind of parse tree for further processing. A Tomita parser would typically build a parse forest, that is, a set of parse trees, where some additional analysis would then select the valid parse interpretation of the statement according to the semantic of the language. See [113] for a complete formalization of such tree-selection filters.

This example can be extended, specially when dealing with C++-specific features, such as references:

```
p ** q;
p & q;
```

Given the counter-example of C, some readers might be tempted to believe that virtually no real-world language complies to this classical model. It is not the case. Pascal [51], Modula II [52], YAFL [31], and Oberon [53] [13] can be parsed using entirely independent lexers and parsers. In the source code of the Oberon compiler which is included in [13], the lexer only knows about identifiers. This is of course not to say that parsing C and C++ cannot be automated at all: [112] describes a DURA-based C++ parser and the amount of work which was added manually for the parser to work reasonably well.

8.4.3 Other languages

We have now versions of RainCode for numerous languages, some of which are barely known, and we believe we have had to solve most of the problems one can encounter when parsing any real-world language. Among the most exotic ones, APS was definitely not the easiest to parse.

APS[11] is an old fashioned fourth generation language, which generates intermediate COBOL code as part of its compilation process. APS is a rather straightforward and well-structured language, but which requires ad hoc techniques because of the semantic meaning it gives to white space, and more specifically, to indentation.

Figure 8.12 APS source sample

The original example:

```
 ...
MOVE 3 TO A
WHILE A > 2
  PERFORM GET-VALUE
  IF (A > 2)
    DISPLAY "Value out of bounds"
    DISPLAY "Please reenter"
    IF (SUPER-USER = 1)
      DISPLAY "Hint: MAX=2"
   ELSE
    DISPLAY "Ok"
  PERFORM DO-CLEANUP
PERFORM PROCESS-VALUE
```

The same example, left-factored:

```
 ...
MOVE 3 TO A
WHILE A > 2
  PERFORM GET-VALUE
  IF (A > 2)
    DISPLAY "Value out of bounds"
    DISPLAY "Please reenter"
    IF (SUPER-USER = 1)
      DISPLAY "Hint: MAX=2"
     ELSE
      DISPLAY "Ok"
      PERFORM DO-CLEANUP
      PERFORM PROCESS-VALUE
```

For instance, the original program in figure 8.12 shows how IF and WHILE statements define their enclosed statement lists by indentation level. The body of the loop – that is, the statements that must be executed as long as the condition of the loop yields TRUE – is defined by the subsequent statements that are indented relatively to the WHILE statement header.

As such, APS clearly is not context-free. Given some form of preprocessing that

[11] The language is so old, there seems to be no online reference for it !

would replace the leading white spaces by ad hoc tokens, it is context-sensitive. It matches one of the examples of context-sensitive languages described in .

In order to deal with this problem, we used an approach in two phases:

- The input is first recognized in left-factored fashion, ignoring the impact of identation level on syntactical constructions altogether. Figure 8.12 shows how the parser would (faultly !) recognize a sample example after left factoring.

- A second pass can then take the parse tree, consider the positions of the various non-terminals (See page 349) and use the various generic tree manipulation facilities (See page 356) to rebalance the tree according to the exact semantics of the language.

This approach in two phases works fine, and is very typical of quite a few parsing problems, where one recognizes an extension or a variation of the original language at the syntactical level, and implements additional checks and corrections at the semantic level.

This technique was developed as hoc for APS, and proved amazingly simple (less than a hundred lines of code) using the wealth of primitives provided by the DURA library to manipulate trees. It soon appeared that it could be formalized, and if well-understood, it could help us cope with numerous other problems.

The idea is simple: when facing an ambiguity problem (as nesting levels in APS), one can recognize a *normal form* (left-factored in the case of APS) so that parsing remains simple and efficient. A separate pass can then take the resulting parse tree, and alter it to make it comply to the exact semantics of the language.

This process is somehow similar to what [113] proposes, except for the fact that instead of recognizing *one* form for the input and change it afterwards to correct it according to the definition of the language at hand, it recognizes *all* forms at parse time, and then uses filters to extract the – hopefully unique – valid parse tree from the so generated parse forest.

Our claim is that the difference between our *rebalancing* operation and the *selection* operation as explained in [113] is neglegible in terms of complexity. On the other hand, by building a single parse tree rather than a potentially large set of interpretations of the input – consider, for instance, and APS procedure with a dozen IF statements for which nesting is controlled by indentation level – the performance of our approach is better by several orders of magnitude since the number of interpretations of the input grows exponentially.

This technique can be used to solve context-sensitive issues, such as significant white-space or nested variable declarations in COBOL. It can also cope with dynamism issues, such as user-defined operators as in Prolog [50] and Algol68 [114]. The problem here is more practical than theoretical: in order to deal with these user-defined operators at parse time, one would have to alter the transition tables of the parser on the fly to adapt to these new operators with their user-defined precedence, which would be utterly unpractical. We would typically recognize these operators with a “standard” precedence and associativity, and rebalance the expression trees afterwards, assigning each user-defined operator usage to its declaration (and consequently, precedence and associativity) to be sure to take all scoping rules of the language into account.

8.4.4 Concluding remarks on `RainCode`

Quality in software development is based on a major assumption one must accept as an act of faith: producing *technically better* software yields benefits which will ultimately turn into measurable advantages. Improving structure, readability and abstraction level will deliver better software to the developer, and ultimately, to the end-user.

To us, `RainCode` has been a way of putting this hypothesis under scrutiny. We now have a complete chain, from the implementation language to the delivery of usable software for end-users who don't care about how it's done, as long as they can use it.

- YAFL's garbage collector delivered stable and controllable memory consumption. Of course, with the recent advent of Java, the virtues of automatic memory management are clear to most of the industry today, but when we started using YAFL almost 10 years ago, we had to fight the idea that manual memory management would provide better overall results.

- YAFL's portability level through intermediate `C` code generation allows us to make `RainCode` available on all platforms, including exotic things such as IBM MVS. The YAFL language and compiler address all the portability issues. This must be compared with the extensive work required to make sure that a `C++` program is portable. Such an effort usually goes through trial and error, actually porting the software regularly to ensure it works on all target platforms, and gradually restricting the usage of the features to a sane subset well-supported by all compilers – in the area of templates, for instance.

- The expressivity of `DURA` , together with its wealth of built-in features – automatic tree building and hierarchy handling, error recovery – has allowed us to produce versions of `RainCode` for new languages in a matter of days. A major factor toward productivity is the way ambiguities can be dealt with: we ignore them, letting the parser backtrack if required. Except for extreme cases such as depicted above for `PL/1`, the performance penalty is negligible.

- The documentation system, while naive in some respects, has proven to be usable and the fact that it can be regenerated automatically in such a way that users can always refer to an up-to-date documentation far outweighs its restrictions.

- The very high-level of abstraction reduces the amount of source code to maintain. For instance, `RainCode` for Ideal (described below when describing the performance of `DURA`-generated parsers) is made of less than 3000 original – understand: not generated – source lines. We can maintain the versions of `RainCode` for all supported languages quite conveniently, bugs can be corrected in a very timely way, and new features can be integrated easily.

- We concentrated on *product-centered* quality, rather than on the processes used to deliver the product. In other words, the development was almost anarchical, poorly formalized in terms of its processes, how steps had to be performed in sequence, etc. On the other hand, we concentrated on ensuring that the end product matches a number of quality requirements we consider of paramount importance

– stability, performance, reuse, structure and maintainability. Besides structural and source code related issues, the only factor toward quality we have set up is comprehensive regression test suites for COBOL and for Java, which are run systematically before committing a new version.

8.5 Performances

In compiler design, and even more, in source code manipulation tools such as Rain-Code, the performance of the parser is not as important as it used to be. A decade ago, the lexer was considered as the probable bottleneck of most compilers [54]. Nowadays, as shown in [57], a compiler such as gcc [81] spends most of its execution time in mapping optimal trees at the code generation level. Modern code generation techniques [115] are often based on np-complete[5] algorithms. In DURA's case, performance has always been considered as just one of the factors that had to be taken into account. Issues such as the classes of language we can deal with, ease of use, quality, etc. have been considered as well, and it is true that such issues are far harder to measure than mere execution times.

In fact, even comparing performances of generated lexers and parsers can be tricky. For instance, comparing DURA and yacc raises several non-trivial issues:

- The performance of the target language (C for yacc, YAFL for DURA). It is clear that by its higher level of abstraction, YAFL is intrinsically slower than C, thereby biasing any fair comparison between the underlying parsing techniques. Rekers [80] raises the same issue when comparing the performance of yacc and his Lisp implementation of GLR.

- yacc produces just a recognizer, while DURA generates complete parse trees automatically, at the expense of some processing power. The DURA-generated parsers don't support the ability *not* to produce the parse tree, making the evaluation of its isolated cost very hard. Rekers [80] cites a factor 2 between a plain recognizer and the corresponding parser that builds the parse tree, but extrapolating this figure to DURA-generated parsers is a bit risky, given the possible differences between the implementation and strategies used when building the parse tree.

- Some languages such as ObjectZ simply cannot be analyzed by yacc-generated parsers, making the comparison impossible. Restricting the comparison to simple cases such as Pascal or an expression evaluator would not make any sense, since DURA has been developed with complex and poorly formalized languages in mind.

It is also true that the real-world parsers we have now are not isolated products, they are usually integrated with some form of preprocessing, semantic analysis and other housekeeping tasks. These additional tasks somehow bias our performance measurements which are based on raw process running time, without isolating syntactical analysis from the rest. All in all, we generally consider that a parser produced by DURA and `lexyt` is 3 to 5 times slower than the equivalent parser generated by `yacc` and `lex`, as far as such a comparison is relevant, that is, if the language to parse can be expressed using a grammar that complies with `yacc`'s limitations: LALR and no lexical ambiguity.

On the other hand, parsers generated by `DURA` seem to compare rather favorably with GLR [66] parsers for similar problems. In [116], the `COBOL` parser generated for [117] is said to be between 5 and 10 times slower than its equivalent `DURA` counterpart[12]. It is also true, however, that this parser was developed with source code renovation only in mind, for which performance is usally not a critical issue when compared to flexibility and versatibility.

Table 8.1: Performance summary

	ObjectZ	COBOL	Ideal
Lines/sec	664	1555	1484
Non-terminals/sec	2909	4252	7327
Operations/sec	44516	29633	23264
Operations/NT	15.33	6.97	3.18
NT/Line	4.371	2.735	4.936
Negative operations:	17.0%	21.3%	1.7%

The relative performance of `DURA`-generated parsers for different languages can be compared fruitfully, and we can almost build a typology based on the performance measured when parsing them. Table 8.1 summarizes results for three such parsers on a PC with a 266Mhz Pentium running Windows NT. Since the parsers are essentially CPU-bound, the operating system is likely to be vastly irrelevant in terms of orders of magnitude. Beside, the results are biased by preprocessing and at least a basic form of semantic analysis for all three of these parsers, but we assume that the impact on performance is comparable in the three cases.

ObjectZ is a language with a very large number of different precedences for operators. Therefore, reducing a literal or a simple identifier into an expression can require over a dozen intermediate reductions. This explains the rather low number of terminals reduced by seconds, since intermediate reductions do not translate in as many non-terminals. Similarly, since many operations – understand, shift, reduces and their negative counterparts in case of backtracking – are required for every non-terminal, the number of operations is relatively high, when compared with `COBOL` and Ideal where non-terminals are created and initialized more frequently.

Quite obviously, the ObjectZ parser would benefit from a less conservative precedence handling than what `DURA` provides (See method *DuraFlat.FlatRule.HandleAssociativity* on page 492). `yacc` (as well as the parser generator depicted in [45]) handles precedence and associativity at the automaton level, by preferring a reduce over a shift in some cases, while `DURA` handles it by transforming the grammar at the rule

[12] This is just a rough estimation, since even the supporting hardware was not equivalent, and in this time of ever increasing processing power, it is of course a major issue when comparing performance.

level, introducing one intermediate grammar rule – understand: one additional reduction – for every precedence level. At some stage, we might consider integrating a yacc-like scheme in DURA , but it is not entirely clear at this stage how it would interact with a backtracking parser, even if the BTYACC [60] experience seems promising.

The COBOL parser measured in table 8.1 is the "universal" COBOL parser on which RainCode relies. It supports a wide variety of COBOL dialects, and supports numerous extensions when compared to what a plain COBOL compiler recognizes. For instance, while it includes a full preprocessor that can expand copy books[13] exactly as any other COBOL compiler would, it can also parses COBOL source files without expanding them as long as they are used in a reasonable way, that is, containing entire variable declaration or statement sequences. This parser also recognizes embedded SQL [118] statements as native statements without going through a preprocessor. Such extensions introduce additional levels of ambiguity: commas are considered as white space when found in plain COBOL, and must be recognized when found in embedded SQL statement. The lexical definition for our COBOL parser includes a lexical definition of the form "Comma ,", that can backtrack to be ignored by virtue of another definition of the form "_Skip ,". In other words, whenever a comma is found in plain COBOL, rather than being ignored and skipped directly at the lexical level, it will first return a Comma code which does not make any sense outside an embedded SQL statement, and backtrack to a skippable token before the parser can proceed with the following token.

Such examples explain why the COBOL parser has such a high rate of negative operations – unshift, unreduce and lexical backtracking – when compared with the other languages in table 8.1. The high performance in terms of line per second can be explained by the verbosity of the language, and the small average number of non-terminals built when parsing a line.

Ideal is a fourth generation language which name is probably some form of private joke. However, it is rather convenient to parse. The very low rate of negative operations indicates that it is likely to be LALR or even LL, after some twiddling here and there. The grammar is rather simple, as indicated by the small number of operations required to synthetize a non-terminal. The low number of operations per second is a consequence of this: since fewer operations lead to a new non-terminal, there are more of such creations of non-terminals, and less time for LR operations.

[13] The copy books are the file inclusion mechanism supported by COBOL.

Chapter 9

Related works

Given any natural number n, there is a way this chapter can be made as large as n pages. Parsers, languages and automata are subjects where hundreds if not thousands of contributions have been made over the last 50 years.

I tried to concentrate on those of the references that are *relevant* to the DURA-lexyt couple. In most cases, I mention *tools* with experimental results, and include more theoretical references only if they add something significant to the issue at hand. DURA and lexyt are based on simplistic theoretical foundations, and it is assumed that all their added value lies in their practical usability. Therefore, there is little interest in enumerating the myriads of alternate parsing algorithms that have been published in the last 30 years.

9.1 Lexical analysis

One cannot consider the issue of lexical analysis without mentioning at least lex [40], which is commonly coupled with yacc to produce full working parsers.

lex typically produces *interpreted* lexers, that is, lexers which, given a transition function implemented as a number of arrays, delivers a finite state automaton. lex produces these arrays as initialized C data structures.

lex also introduces a level of flexibility, by allowing the user to determine the action that must be taken whenever a token has been recognized as a piece of code in the host language (Usually C, but can be C++, Pascal, etc.). This is the only fundamental reason why lex must generate a lexer in the form of a compilable source file: it would have been otherwise possible to produce the transition tables in an external file which would have to be read by a universal lexer to deliver the required behaviour.

lexyt retains lex's mechanical behaviour, that is, the automaton part of the story. However, in order to provide better performance, lexyt generates *compiled* automata, that is, automata where the state transition mechanism is implement by a number of nested CASE statements.

lex supports right-context disambiguation as shown in figure 2.5 in page 24. lexyt does not support it for several reasons:

- It is not as necessary in `DURA-lexyt`'s context, since backtracking always allows to reconsider an erroneous choice in case of conflict

- It makes the lexer generation process far more complex

- The generated lexer is not guaranteed to have linear performance any longer. In the worst case, the right-context extends to the end of the input on every token, and the lexer's behaviour is quadratic.

Flex++[1] is just a thin OO layer around the freely available reimplementation of `lex` named Flex, and which encapsulates functions in a class.

GLA [55] and Re2C [56] produce highly tuned lexical analyzer directly in `C`, using highly tuned `switch` and `if` statements to control the logic, rather than actually implementing an automaton as states with transitions as `lex` and derivatives do.

9.2 Syntactical analysis

9.2.1 `yacc` and followers

We'll start our guided tour of compiler-compilers for context-free languages with `yacc`, which is the absolute reference tool for this domain.

`yacc` can be characterized as follows:

- Its input format is roughly equivalent to `DURA`'s flattened representation (See module *DuraFlat* on page 474).

- It can produce a parser for LALR(1) [73] [74] languages, extended by a rule to solve conflicts, namely, preferring shifts over reductions in case of a shift-reduce conflict; and preferring the first occurring rule in the specification in case of a reduce-reduce conflict. `DURA` will also first attempt to shift if applicable, but will also try all applicable reductions in sequence until one eventually succeeds.

- `yacc` does not support lexical backtracking in any form.

- `yacc` manages associativity and precedence at the *automaton* level rather than at the grammar level, as `DURA` does (See method *DuraFlSpec.FlatSpecification.HandleAssociativity* on page 574).

- `yacc` – at least in its original incarnations – produces table-driven parsers in `C` only.

- `yacc` does not include any tree-building capability. This implies that a parse tree, if any, is built by providing ad hoc code in the semantic actions attached to grammar rules. On the positive side, this also implies that `yacc` can be used in circumstances where building a parse tree would be unpractical, as for command-line interpreters, for instance.

[1] No reference to any usable piece of information about Flex++ has been found.

- `yacc`'s error recovery scheme is basically comparable with `DURA`'s, and this is not meant to be a compliment for any of the two tools.

Based on `yacc`, several directions have been investigated. There are now versions of `yacc` (or equivalent) for other languages than `C` only (Ada [119], Pascal [120] [121], Java [122]). Bison [123] is a reimplementation of `yacc` which is freely available through a GPL licence (AT&T holds the original rights on `yacc`).

Then, other extensions based on `yacc` improved it in more fundamental ways. For instance, at least a freely available version named YAY [124] as well as a commercial implementation made available by MKS [120] can produce parsers for LALR(2) languages. Lemon [125] addresses a minor but annoying problem about `yacc`, namely, its lack of robustness when it comes to memory deallocation during error recovery. When performing error recovery in plain `yacc`, entries of the LR stack are discarded brutally, even if they contain pointers to memory which cannot be reclaimed any longer. It is commonly advised to use conservative garbage collectors (such as [126] [127]) to address this problem in plain `yacc`[2]. While retaining most of `yacc`'s concepts, Lemon's input syntax is not strictly `yacc`-compatible. It is also different, in the way the parsers it generates are driven by the lexer while most other parser generators (including `DURA`) work the other way around: the parser leads the game, and calls the lexer whenever it needs a new token.

ZYacc [128] provides a more fundamental feature in order to deal with reduce-reduce conflicts: one can attach *validation code* to a non-terminal. Before performing a reduction, the validation code attached to the corresponding grammar rule is executed, and the reduction is cancelled if this evaluation yields 0, thereby enabling another rule – which would have not been executed at all in plain `yacc` due to its reduce-reduce strategy which inconditionally selects one of the rules to reduce. A similar feature could be added to `DURA` in the future to cope with the `PL/1` dialect issue depicted in 8.4.1 in page 936.

The following example, extracted from the ZYacc documentation, shows how semantic tests can disambiguate a situation where distinguishing a function identifier from an array identifier at parse time is essential in absence of any backtracking mechanism.

```
exp
  : fnID '(' indexExpList ')'
  | arrayID '(' fnArgList ')'
  ;
fnID
  : ID_TOK($id) %test(idKind($id) == FN_KIND)
  ;
arrayID
  : ID_TOK($id) %test(idKind($id) == ARRAY_KIND)
  ;
```

[2] This issue is of course totally irrelevant with regard to `DURA` since YAFL includes an automatic garbage collection system.

Visual Parse++ [121] is a commercial product which produces parsers in C, C++, Pascal (or more accurately, its commercial implementation Delphi [129]) and Visual Basic. It uses a variant of LALR, and produce monolythic parsers and lexers. It does not build a parse tree automatically, which must be built using user-defined semantic actions. Visual Parse++ also includes various debugging and 3D visualization facilities to monitor a parser while running, and better understand its behaviour.

The latest version of Visual Parse++ also includes the ability to produce a GLR parsers as described in [66] to handle natural language processing. It does not support any kind of backtracking at the lexical level.

Of all the efforts somehow directly derived from `yacc` or inspired by it, BTYACC [60] is probably the one `DURA` can best be compared to. BTYACC stands for backtracking `yacc`: whenever a conflict arises, the parser freezes the current state, and executes the various actions that can be taken in sequence until the ambiguity is lifted. In trial mode – that is, when parsing to lift an ambiguity – some of the semantic actions can be inhibited, to prevent the system from being altered irreversably while investigating a parsing path that might not be the right one.

Quoted from the BTYACC documentation, the following explanation:

> *BTYACC is a modified version of* `yacc` *that supports automatic backtracking and semantic disambiguation to parse ambiguous grammars, as well as syntactic sugar for inherited attributes (which tend to introduce conflicts). Whenever a btyacc generated parser runs into a shift-reduce or reduce-reduce error in the parse table, it remembers the current parse point (*`yacc` *stack and input stream state), and goes into trial parse mode. It then continues parsing, ignoring most rule actions. If it runs into an error (either through the parse table or through an action calling YYERROR), it backtracks to the most recent conflict point and tries a different alternative. If it finds a successful parse (reaches the end of the input or an action calls YYVALID), it backtracks to the point where it first entered trial parse mode, and continues with a full parse (executing all actions), following the path of the successful trial.*
>
> *Actions in BTYACC come in two flavors – -actions, which are only executed when not in trial mode, and []-actions which are executed regardless of mode. There are also inherited attributes, which look like arguments (they are enclosed in "()") and act like []-actions.*

Clearly, just as `DURA`, BTYACC aims at parsing *locally ambiguous languages*, where a local non-determinism can be solved by considering a more of the input, as opposed to *globally ambiguous languages*, where the input can yield more than one parse tree. Error recovery is also similar: BTYACC also remembers the most advanced position in the input to start recovering at that stage.

BTYACC is also similar to `DURA` in the sense that its backtracking facility is *automatic*, while Lark's [63] and ANTLR's [64] backtracking is controlled by the programmer, who must indicate how to explore paths before taking a decision about the proper parsing action to take.

Despite the similarities, `DURA` and BTYACC are also very different in some respects:

- Notation. BTYACC accept the standard `yacc` notation, and grammatical constructions such as repetitions and optional components must be translated manually into the flattened representation, while `DURA` performs such translation automatically.

- Building the parse tree. Building a parse tree can only be achieved using user-defined semantic actions, while this process is automated in `DURA`.

- Performance. As long as there is no conflict, a BTYACC-generate parser's performance is roughly equivalent to `yacc`'s, which is expected to be significantly better than a `DURA`-generated parser. We expect the behaviour in case of conflict – that is, when backtracking must be used – to be similar to `DURA`'s, except for the intrinsic differences between a parser in `C` and the equivalent YAFL implementation.

  The claim about BTYACC's performance to be equivalent to plain `yacc`'s in absence of conflict must be taken with a grain of salt: when facing a conflict, `yacc` behaves deterministically (always prefer the shift over the reduce, and the first reduction in the specification if more than one are applicable). This property is commonly used by grammar writers, who consciously take advantage of the automatically selected behaviour.

  Even if the `yacc`-selected default behaviour when facing a conflict leads to a valid parse tree, BTYACC will start its backtracking machinery, thereby inducing a performance overhead when compared to the corresponding plain `yacc` parser.

- Lexical backtracking. BTYACC sticks to the classical model, as depicted in figure 2.2 in page 21. It is based on the assumption that the input can be unambiguously divided into lexemes. This clearly shows in the way CobolTransformer [102] – which is implemented using BTYACC – manages different `COBOL` dialects. The set of keywords must be selected externally, rather than leaving the work of guessing what is a keyword from what is a user-defined identifier to the parser. Of course, for all practical cases when dealing with `COBOL`, the solution provided by CobolTransformer is perfectly adequate. However, it shows an intrinsic limitation of BTYACC-generated parsers: they cannot parser languages such as `PL/1` and `FORTRAN90` where lexical backtracking serves a more fundamental purpose. These languages don't match the classical model, and no a priori selection of keyword can trick a parser into recognizing them accurately.

9.2.2 Simpler approaches

COCO/R [130] [131] is a tool developed by Hans Peter Mössenböck at the ETH in Zurich, in the true spirit of Modula II and Pascal: stick to the bare necessities, and deliver a small, concise and efficient tool. It is the other end of the spectrum: in some respects – such as error recovery, for instance – it is limited, even compared to primitive tools such as `yacc`, but given its scope of application, it works very well.

COCO/R includes both a lexer and a parser generator, and can generate Oberon [13], Modula II [52] and Delphi [129] code. The lexer generator takes regular expressions, and produces automata. The code generation scheme is closer to `lexyt` than to `lex`: the transition table is turned into a `CASE` statement over the states, rather than by interpreting initialized tables[3].

COCO/R produces recursive-descent parsers, and makes use of the error recovery method described in [132], which is simple and efficient, but, which requires more insight from the developer than less tuned and more automated approaches.

In [130], Mössenböck clearly describes the scope of his work, by claiming that there is "*little reason why a programming language should not be designed in a way that makes it amenable to single-pass compilation and top-down parsing*". As a matter of principles, I fully agree with this statement. It also shows how `DURA` and COCO/R basically address different issues: `DURA` is not only about parsing newly designed language. It must be able to deal with existing code portfolios, no matter how poorly designed their programming languages originally were.

9.2.3 More integrated approach

SableCC [133] [134] is a compiler generation framework that can be compared with `DURA` in some respects: it generates plain LALR(1) parsers, and, unlike `yacc` and alikes, builds typed parse trees automatically. When given a grammar, SableCC produces three sets of Java classes:

- A parser class, that implements the actual parsing logic
- A number of classes to describe a parse tree. These classes mimic the grammar rules directly.
- A visitor class, inspired by [84], that can be used to walk through the parse tree, *intercept* nodes depending on their class. This class can be derived and include specific action to be taken whenever a node of a given class is encountered in the parse tree.

It is of course vastly similar to `DURA` in its ability to produce the parse tree automatically. The major difference lies in the way SableCC *generates* the parse tree classes, thereby inhibiting the insertion of any user-defined semantics in them and requiring and external visitor class to plug intelligence in the parse tree. `DURA` uses user-defined classes *extended* with grammar information to produce a parser. These non-terminal classes can include any intelligence without having to resort to external visitor objects.

Cocktail [99] is a compiler-compiler framework that goes from lexical analysis to code generation, including several parser generators (LL(1), LALR(k) and backtracking LR). Cocktail has been used to produce several production-level compilers and translators.

Cocktail used to be freely available [99] but its latest developments have moved to the commercial world lately [135],

[3]This design is clearly superior to `lex`'s table-driven approach in terms of performance. However, it is possible that performance was not the sole motivation behind this: Oberon and Modula II don't provide the ability to declare initialized tables as `C` does, making the interpreted approach less convenient to implement. Not surprisingly, a similar reasoning baszed on YAFL's inability to declare initialized tables led to the same design decision for `lexyt`.

Cocktail now includes a backtracking LR(k) parser generator named *Lark* [63], aimed at replacing the LALR(k) parser generator which performance advantage over a more generalized approach gets less relevant as time goes by.

Lark can produce parsers in numerous languages: `C`, `C++`, Ada, Modula II and Eiffel [86]. Lark supports backtracking, but in a more primitive form than `DURA`or BTYACC in the sense that it is the user's responsibility to indicate how to lift ambiguities. At some point, the successful or unsuccesful parse of a grammatical expression – referred to as *syntactic predicate* – given by the user determines whether a given parsing path is to be investigated or not. Backtracking, in Lark's context, is restricted to the ability for the parser to return from the trial parsing of a piece of input using a syntactic predicate.

The CoCoLab web site [135] refers to a `PL/1` parser, generated using Lark. As seen in figure 2.4 in page 23, `PL/1` cannot be parsed naively using the classical model as depicted in figure 2.2 in page 21.

Since Cocktail does not seem to support lexical backtracking in any form, I'd speculate that the `PL/1` parser described in [135] uses an entirely different technique: since Lark can deal with locally ambiguous grammars, it can include a grammar rule of the form:

```
UserDefinedId :- BEGIN |
   END |
   IF |
   DECLARE |
   DO |
   ...
   OTHER_ID;
```

and all usages of a user-defined identifier refer to "`UserDefinedId`" for which the grammar explicitely states that a user-defined identifier can be any of the `PL/1`'s reserved words, letting backtracking solve ambiguities if any. In other words, this hypothesis is based on the assumption that the lexer used in this `PL/1` parser is a plain, non-backtracking one.

ANTLR [64] (Previously known as PCCTS) is a parser-generation tool which unifies various concepts in a common formalism:

- It includes a LL(k) parser generator
- It takes advantage of the fact that regular languages are a subset of context-free languages to use the same tool – on a different alphabet (See page 13) of course – to produce lexical analyzers
- Pattern matching tree manipulators can also be written using the same tool.

ANTLT uses the technique described in [136] to avoid the exponential explosion a naive implementation of LL(k) leads to. It generates parsers in `C++` and Java, and its own Java source code is freely available. When generating `C++`, ANTLR produces reference counting code to address the absence of garbage collector.

In addition, ANTLR supports two features aimed at coping with the limitations of LL(k) and, more generally, *pure languages*, that is, languages that can be parsed using an attribute-free grammar:

- Semantic predicates, where the user can provide a snippet of code in the host language to allow or exclude access to a grammar rule
- Syntactic predicates, where the user instructs the parser to *try* to parse a given grammar expression starting at the current position in the input and depending on its success or failure, allow or exclude access to a grammar rule. It is thus a form of user-controlled backtracking, similar in all respects to what Lark provides.

The parse trees generated by ANTLR are not typed, they are made of nodes of a single class where access methods must be used to discriminate node types. `DURA` and SableCC generate fully typed parse trees.

ANTLR grammars are monolythic, but support a composition primitive named *grammar inheritance* where a grammar can be defined as a variant – extension, redefinition – of an existing reference grammar.

JavaCC [65] is essentially similar to ANTLR, sharing most of its advantages and drawbacks:

- It generates LL(k) recursive descent parsers
- It provides semantic and syntactic hooks to solve cases where LL(k) is not powerful enough
- It includes a tool named *JJTree* which is used as a preprocessor on the JavaCC input to include AST generation code.

9.2.4 Generalized parsing methods

Generalized parsing methods can handle any context-free grammar, and are not restricted to any LALR, LL or other class of grammar. There are four commonly referred to methods in the bibliography:

- Unger [137]
- CYK by J. Cocke, D.H. Younger [138], T. Kasami [139], also known as *chart parsing*
- Earley [30]
- Tomita [66]

Unger and CYK are mentioned for historical reasons only: they have rather poor behaviour – cubic in all cases – required large data structures, and, at least in the case of CYK, required that the grammar is first put in a form of which CNF (Chomsky Normal Form) is a subset [140] for a reasonably simple parser to be generated. Except for being slightly easier to understand, Unger and CYK hold no practical advantage over Earley and even more so, over Tomita, which delivers better performance in virtually all cases.

9.2.4.1 Earley

An *Earley item* is based on an LR item as defined in the $FlatItem$ (See page 623) class. An LR item is a grammar rule, together with a position within the grammar rule. At every position in the input, the algorithm maintains a set of Earley items, which indicate all the grammar rules currently being analyzed, and the current position within each of these grammar rules.

In addition, each Earley item also maintain a position that indicate the position in the input stream where the analysis of the attached grammar rule started.

An Earley parser [69] [30] starts by attaching an initial set of Earley items to the initial position in the input stream, and builds the set of items for every position using three functions: the *predictor*, the *scanner* and the *completer*.

If, for any position in the input stream, the attached set of the Earley items is empty, the input is not a valid production of the grammar. For a more formal treatment of Earley parsers, see [30] or [70][4].

For instance, given a grammar rule of the form:

$$IfStatement\ ::=\ "If"\ Expr\ "Then"\ Stmt\ "Else"\ Stmt$$

an Earley item such as: $"If"\ Expr \bullet "Then"\ Stmt\ "Else"\ Stmt, 51$ (where $\bullet$ denotes the position of the item within its enclosing right hand side) in the item set attached to position 88 indicates that when facing the 88th token of the input, we might be currently recognizing a $IfStatement$ that starts in position 51 of the input, and this 88th token must be a $"Then"$ for this item to have a successor in the item set attached to position 89.

Earley parsers are cubic in the worst case, quadratic when the language is ambiguous and quasi linear in all other realistic cases. Beside, it does not put any normal form restriction on the way the grammar is built, except for grammatical cycles.

Simple observations [70] have shown that one of the drawbacks of Earley parsers lies in their tendancy to produce large sets of Earley items out of which only a small fraction produce a possible interpretation of the input. Hence, lots of efforts have been invested in trying to reduce the number of new items added to the current state at each stage of the analysis.

One of the simplest way comes directly from the world of LR parsers: Earley parsers can be improved by using a lookahead string, which determines whether the completion – which is the Earley equivalent of reduction – is valid or not. In [67] and [68], a variant of Earley named BPS (Bouckaert, Pirotte and Snelling) is presented: instead of waiting until the end of a rule to decide whether a reduction can be performed using *reduction lookahead*, BPS uses *prediction lookahead*, where the same technique is used to decide whether it is worth the trouble to consider maintaining the Earley items for a given grammar rule at the beginning of the right hand side, rather than at its end.

Despite their apparently nice properties, Earley parsers have not been used a lot compared to other, less powerful parsing methods. Marpa [142] is small parser-generator

[4] This excellent general reference about parsing technology has been out of print for a while, but its text can be downloaded from [141].

that can be used in as an extension of Tcl [143] to embed small parsers in programs. Flexibility was considered more important than performance: the user does not have to worry whether adding a rule retains the LALR (or LL) property of the grammar.

Most other references to Earley are related to natural language processing, as this algorithm shines when facing languages far more ambiguous than most programming languages, as shown in [80].

9.2.4.2 Tomita (GLR)

Tomita [66] (or GLR) parsers are based on previous work by Bernard Lang [144]. They are easy to understand:

- A GLR parser is based on a plain LR parser [59], but instead of holding a single LR stack (the current state being induced by the top of the stack), it holds a *set of LR stacks*, initialized as a singleton holding a single empty LR stack.
- For every input token, let every LR stack in the set evolve as atomic LR stack would when given the token.
- In case of conflict, that is, a LR stack for which more than one action can be taken – shift or reduce, duplicates of the LR stack are added to the set of stacks, reflecting all the possible actions.
- All the stacks for which the current input token does not lead to a valid transition are removed from the set of active stacks.
- If, at one stage, the set of active stacks is empty, it means that the grammar cannot match the input in any way, and the parser fails.

If implemented naively, cloning the LR stack every time a conflict arises would be very costly. In his original paper, Tomita proposes an alternate implementation of the stacks, as linked lists of stack slices. Cloning a stack simply means allocating a new stack slice that points to a stack prefix it shares with others.

When two different paths evolve to a common configuration (LR stack and LR state), the two LR stacks are merged, achieving a DAG structure, reflecting the semantics of sets (which cannot contain an element with a cardinality greater than 1) as used in the informal description above.

Tomita stated that this algorithm would work for all cycle-free context free languages, but he overlooked *hidden left recursions*, as shown in the case of `DURA` (See page 251) and as shown originally for Tomita's algorithm in [76]. This is of course no coincidence: it is clear, after analyzing the behaviours of the two algorithms thoroughly, that a `DURA` parser is just the single-threaded depth-first walkthrough of a Tomita parser DAG.

One of the major advantages of Tomita lies in its ability to build reasonably compact parse forests while parsing. `DURA` does not need this ability, since it aims at inputs for which at most a single parse tree can be produced. Tomita is based on an optimistic assumption, namely that large parts of the input can be analyzed with a plain

LR parsers, without requiring LR stack cloning. Practical grammars are seldom ambiguous enough to justify the heavy machinery required by an Earley parser. In [66], Tomita claims that this algorithm produces parsers that are 5 to 10 times faster than the corresponding Earley parser for realistic grammars. Rekers [80] seems to support this statement based on experimental data, and indicates that Earley would behave better on excessively ambiguous grammars only.

Another experience of using Tomita parsing for programming languages can be found by Wagner and Graham [145].

`DURA` takes this optimistic view on the world even further: not only can a plain LR parser handle most of the input, but in case of conflicts, we won't have to go very far to bump into a problem if we select the wrong path.

9.2.4.3 The real drawback of forward only generalized parsing methods

In his original paper, even Earley himself acknowledged that this parsing method, while quasi-linear in many realistic cases, would see its performance controlled by very large constant factors. In practice, however, static computation of sets of Earley items (just as `yacc` and `DURA` statically compute sets of LR items) can lead to performance levels which are roughly comparable with LALR parsers [146].

Similar optimizations have been applied to GLR parsers [147], at the cost of increasing the number of grammar rules significantly.

Given these efforts, one can wonder, how come we still use or even consider LALR? If the performance is roughly the same, why do we still bother using tools that require a major understanding of the restrictions imposed by the class of language to be used proficiently?

I believe the reason goes far beyond issues related to classes of grammars only. I think it has to do with the ability to trigger actions when given constructs have been recognized. Both Earley and Tomita parsers only support forward movement on the input, and consider the various possible interpretations of the input simultaneously. Hence, adding some form of user-defined processing while parsing is cumbersome: if some context data structure must be used, it must be replicated whenever an alternate path is considered. Then, when two different paths are reconciled in a a single state, the corresponding data structures must be merged, and one can think of examples where this merge operation si simply not possible.

Earley and Tomita parsers typically deal with *pure* languages, that is, languages that can be analyzed successfully using a grammar, and *a grammar only*. As shown in 8.4.2 in page 938, `C` and `C++` are not pure according to this definition. `C` and `C++` cannot be parsed by a parser generated from an attributeless grammar without any user-defined additional processing.

For instance, considering a language where types are added to a symbol table – as in `C` and `C++` – in a way which is crucial for the successfull parse of the rest of the input, one can imagine a situation where two paths, that is, two different *interpretations* of a part of the input are such that they induce the addition of different types to the symbol table. The context objects attached to these two paths cannot be merged. This is more than just a symbol-table issue: the merge is not valid because the two states are not equivalent: the types known by the system when in these two states are not the same, thereby implying possible different parsing behaviours.

This does not mean that C or C++ cannot be parsed at all using these algorithms: for instance, one can use a Tomita parser to build a *parse forest* for a given C source file, and an extra filter will extract a valid parse tree from this parse forest on a later stage. In other words, the responsibility of building a valid parse tree cannot be left to the parser alone. Somehow, these techniques *weaken* the parser, by in the sense it never refuses a valid input, but it might accept an invalid input, leaving the responsibility to detect it to a latter pass.

Of course, this is a common trend in parsing: we want the parser – specially if generated automatically – to detect as many things as possible, but pragmatically, we defer some of these checks to semantic analysis. For instance, as shown in, detecting ensuring that the closing identifier of a procedure in languages such as YAFL or Modula II requires context sensitiveness, and cannot be handled by context-free parser generators. However, it's all a question of degree: knowing that we'll have to leave some of the processing to semantic analysis does not mean that it does not make any difference whether some or all of this analysis is performed by the parser.

9.2.5 PRECC

PRECC [61] [62] (PREttier Compiler Compiler) is one of the most original contributions to the world of parser generators. Its basics are amazingly simple: each grammar rule is turned into a C function which, starting at a given position of the input, can *succeed* and accept the non-terminal and update the current position in the input accordingly, or fail, leaving the current position in the input unchanged.

Using these basic blocks, PRECC can produce backtracking parsers where the various transitions – implemented as separate C functions – are tested in sequence until one succeeds.

PRECC also provides higher-order operations, which take a grammar rule as parameter. For instance, one can define a generic grammar rule $Params_A$ parameterized on grammar rule A, defined as:

$$\begin{array}{lcl} Params_A & ::= & \text{"("}\ Mult_A\ \text{")"} \\ & | & \epsilon \\ Mult_A & ::= & A \\ & | & A\ \text{","}\ Mult_A \end{array}$$

In terms of the resulting implementation, the C function generated for $Params_A$ will take a pointer to the function for A as parameter.

PRECC can produce a parser for any grammar, as long as it contains no left-recursive grammar rules. This restriction is easy to understand, since PRECC actually generates backtracking recursive descent parsers, which consider left-recursion as a lethal disease.

PRECC and DURA have quite a lot in common:

- The set of grammars they can handle are basically the same

- They both produce backtracking parsers
- They both provide ways of *controlling* the backtracking level, by means of cut and cur-like operators, as available in the backtracking engine of Prolog [50].
- More anecdotically, just as `DURA` was developed with the parsing of ObjectZ as one of its objectives, PRECC seems to be at least partially aimed at parsing plain Z [100] [148]. This does not come as a surprise: one of PRECC's authors, namely, Jonathan Bowen, is also a respected Z author [149].

There are also major differences between PRECC and the `DURA`-`lexyt` couple:

- Left-recursion. PRECC does not support left-recursive grammars, while `DURA` does not put any such restriction on the input format. As a matter of fact, just as `yacc` does and for exactly the same reasons, `DURA` even prefers left-recursion to right-recursion.
- Performance. Since their behaviour is globally comparable in terms of atomic operations, one can expect a PRECC parser to be faster than the corresponding `DURA` parser by avoiding the overhead of the YAFL runtime system when compared to raw `C`.
- Expressivity. The grammars one can express using PRECC and `DURA` are basically the same. `DURA` clearly has a more expressive set of operators, closer to the way grammars are commonly described. On the other hand, higher order operators as supported by PRECC allow more sophisticated grammatical construction, perhaps at the expense of readability, at least for a reader who does not manipulate functional programming concepts on a daily basis.
- PRECC supports *semantic conditions*, implemented as user-defined code, and which are checked by the generated parser before proceeding. This is how PRECC supports parsing of context-dependent languages. `DURA` supports the ability to redefine the `Commit` method (See page 347) which can perform user-defined code, and possibly cancel the reduction and force the parser to backtrack. The different between PRECC's approach – at any time, including even *before* starting to recognize the grammar rule – and `DURA`'s – at reduction time – is typical of the difference between top-down and bottom-up parsers.
- `DURA` builds the parse tree on the fly using a fully typed object model. PRECC does nothing of the kind: it is up to the user to write the tree building code in the semantic actions attached to grammar rules.
- Dealing with lexical ambiguities. PRECC is plug-compatible with `lex`-generated lexers, and holds to its limitations. Alternatively, PRECC can be used to produce a scannerless parser (See 9.2.7 in page 961).

9.2.6 ASF+SDF

ASF+SDF [150] is a meta-language development environment – "meta" in the sense that it can be used to develop language development environments – aimed primarly at software renovation in a way which makes it comparable to `RainCode` (See page 920). It is based on two components:

ASF [151] is an algebraic specification formalism

SDF [152] is the parser generator, based on the GLR [66] parser developed as part of the work of Rekers [80] to support hidden left recursions (See 5.5.4 in page 251). It also supports incremental parser generation in a form somehow similar for GLR parsing to what Horspool [153] [154] proposes for LALR grammars.

[49] introduces two criterias to evaluate parsing technologies, among which ASF+SDF:

Modularity describes the ability to define the grammar in a modular way, that is, as a collection of manageable components. For instance, `yacc` is not modular, while `DURA`, by mapping the grammatical information onto the class structure as supported by plain YAFL clearly is modular.

Compositionality is stronger than modularity: it denotes the ability to compose grammar components in a seamless way. `yacc` can serve as counter example here as well: `yacc` can handle two LALR(1) grammars, but might as well not be able to produce a working parser for the composition of these two grammars, as it might not be LALR(1) any longer. From this point of view, `DURA` is doing reasonably well: except for the two rather rare cases that can cause a `DURA`-generated parser to cycle (See 5.5.1 in page 244), combining `DURA` grammars is safe.

On the other hand, since `DURA` works by referring to an external lexer (typically produced by `lexyt`), combining two parsers might require that one also merges lexers, which is far from being as easy, since `lexyt`-generated lexers are not modular.

However, [49] refers to compositionality in its ability to quickly define specific parsers, for instance, in the context of `COBOL`, defining a parser that does recognize CICS or SQL extensions or not. `DURA` supports this ability fully, as the set of the grammar rules used to build a parser is deduced from the set of modules
YAFL in the corresponding YAFL project. Hence, by defining wrapper projects that do – or don't – import specific modules, one can build project-specific parsers conveniently. This approach is less flexible than ASF+SDF's, since it implies that compositionality is controlled by static compilation of a YAFL project. ASF+SDF is based on incremental parser generation, and can adapt to specific grammar configurations dynamically.

While the YAFL-based approach to compositionality is technically sound, it has not been applied in `RainCode` (See 8.3 in page 920) since its aim is to provide a *unique* parser for all `COBOL` dialects. So far, our approach has proven workable, but in the future, if managing all possible extensions and dialects within a single parser becomes impossible, we might revert to the production of specific versions depending on the set of applicable grammar rules.

In the context of [117] and as shown in some of the examples in [49], ASF+SDF is used as a core technology, that must be tuned to adapt to a specific code renovation project at hand. This bias toward a process (vs. a product, as for `RainCode`) explains why compositionality is considered important in their context.

There is an intrinsic difference between `DURA-lexyt` and ASF+SDF, induced by the underlying parsing technology: `DURA` will always return at most one parse tree for the input. It will perhaps not be perfect, and various generic manipulation facilities are provided to change the returned parse tree using information (dynamic operator precedences, for instance) that goes beyond what `DURA` could handle.

On the other hand, ASF+SDF uses GLR, which can return a set of parse trees structured as a parse forest, which is far more compact than the naive exponential enumeration of all possible parse trees one by one.

In theory, this parse forest serves no practical purpose in case of a programming language, which, unlike natural languages, is not globally ambiguous. In other words, while GLR helps considerably by taking care of local ambiguities, its ability to deal with global ambiguities could be seen as an overkill.

However, Klint and Visser [113] describe a formalism where one can defined *tree filters* can be used to constrain a SDF parse forest to a – hopefully, unique – parse tree. Among possible applications, they mention lifting ambiguities based on precedence. They also somehow weaken the practical applicability of this technique by acknowledging that the formalism might not be better implemented as is, and that some of the filters might be better applied earlier in the process – understand: while parsing.

All in all, the mere ability to maintain forests rather than trees can have interesting applications, but the performance penalty is likely to be significant.

In [49], it is indicated that ASF+SDF is about to move to scannerless parsers, which has been confirmed by [116] with rather promising results.

9.2.7 Scannerless parsers

Scannerless parsing is a technique where the lexical definition is merged into the syntactical definition, taking advantage of the fact that context-free languages are a superset of regular languages. The resulting parser, rather than taking a stream of higher-level tokens as input, deals directly with characters.

Scannerless parsing was first introduced by Salomon and Cormack [155] [156], and has been investigated further by Visser in [157], where he integrates the lexical definitions in a GLR parser. He raises numerous issues in the process:

- While the classes of languages clearly allow the integration of the lexer in the parser, some of the facilities one takes for granted in lexical analysis have to be somehow simulated when plugged direcly in the parser. [157] lists two of such facilities, namely, *Longest Match* which ensures that the longest possible token is returned and the *Prefer Literals* rules, where one can reduce the level of ambiguity by explicitely excluding some of the possible interpretations of the input. Both of these behaviours are naturally required by real world lexers, and are provided by `lex` [40], `lexyt`, and most lexical analyzer generators.

- The *Longest Match* property is implemented using *FOLLOW* set restriction. It is a direct transposition of the SLR(1) [72] reduction constraint: the token is reduced if the lookahead character is a member of its *FOLLOW* set. This approach is correct in most real-world cases, even if examples can be built where it yields wrong results.

- When mapping an existing lexical specification made of regular expression into a scannerless parser, performance is guaranteed to be linear while recognizing a lexeme by an LR automaton. The problem here is that we are dealing with the tightest loop of the entire parser front end. Having a guaranteed linear behaviour is nice, but constant factors are likely to be much bigger in GLR than in a conventional DFA, thereby making this approach, if not unrealistic, at least, costly in terms of performance.

Some other points should be investigated, such as the effect on the number of states of the push-down automaton. Some of the parsers we deal with produce over 2000 states, and multiplying this figure by 3 or 4 would raise hard to solve logistic issues when compiling the generated parsers.

Scannerless parsers are a possible answer to a major issue in compiler design: when the separation of responsibilities between the lexer and the parser proves inappropriate, they integrate all the combined lexical and syntactical intelligence in the parser which has powerful mechanisms to cope with ambiguities.

`DURA-lexyt` take another approach to solve this problem: it keeps the lexer and the parser separate, but adds intelligence – backtracking – in the lexer to make it able to cope with required extra level of complexity and flexibility.

9.2.8 More fundamental work

Parsing Schemata [36] is a theoretical effort to classify various parsing algorithms, and defines the transformations needed to go from one algorithm to another. The firm knowledge of these transformations can then be used to *cross-fertilize*, that is, transform an enhancement to a given parsing algorithm into a similar enhancement to another parsing algorithm, by *passing it thru* the transformation that turns one algorithm into the other.

Klaas Sikkel gives an example of such cross-fertilization, by transforming a classical parallel Earley parser into a similar Tomita parallel parser, taking advantage of the similarities between the parsing schematas of these two classes of algorithms. Even if the performance improvement measured by Sikkel on his “synthetic” algorithm is not very impressive, the theoretical result, obtained by transforming an algorithm into another one by almost mechanical means is a major breakthrough.

In *The Functional Treatment of Parsing* [77], René Leermakers visits the world of parsing using an uncommon and refreshing view, based on functional programming. He describes recursive descent, recursive ascent [158], LR, Earley, Tomita and others, using a compact and efficient functional notation. His claim that this notation is supposedly far superior than the common apprach is debatable, and, in my opinion, depends greatly on how familiar the reader is with functional programming issues[5]. However, he shows clearly how some complex data structures used with various parsing algorithms can be made much simpler by relying on functional programming primitives such as memoing. He introduces a preserving grammar transformation which maps A onto F_A, such that F_A contains no left-recursive grammar rule and no right hand side with more than two symbols. He then shows how parsing A using recursive ascent [158] is equivalent to recursive descent on F_A.

The author's claim that this formalization can be used as a guide for implementation can be challenged as well. Defining a LR state as a function which, given a lookahead string, returns the function that implements the target state is intellectually tempting, and would work fine as long as one deals with correct inputs only. On the other hand, when implementing error recovery strategies, one need the ability to *reason* at a meta-level on the states stored on the LR stack. The translation of states into function as implemented by commonly available programming language would make this reasoning phase very awkward.

These works are both impressive[6] but their aim is no `DURA`'s: they are clearly directed toward natural language parsing, as the examples they refer to extensively raise typical natural language issues, such as *global ambiguities* as opposed to local ambiguities which only require a wider window on the input – the lookahead – to be lifted.

Of course, the question is whether the general parsing algorithms, by being intrinsically almost universal by nature, can apply to "simpler" problems, such as parsing `COBOL` or `PL/1`. GLR [66] is a bit on the egde, and [49] shows how it is vastly used in `COBOL` reengineering tools, while it is also often mentioned for natural language processing.

Despite this apparent evidence, I strongly believe that parsing real-world programming languages raises problems that cannot be handled efficiently by the heavy machinery used to parse natural languages:

- Local vs. global ambiguities. Programming languages can be locally ambiguous, but ultimately, a given source must hold a single interpretation. Building parse forests instead of parse trees serves no practical purpose.

- Input of tens of thousands of words, as opposed to tens of words when dealing with natural languages. Algorithms in n^3 or even n^2 are not acceptable, while they are rather common in natural language recognition.

- Some languages – such as `C++`, for instance, see 8.4.2 on page 938 – cannot be analyzed by *pure* parsers, that is, attributeless parsers with no added intelligence.

[5]Beloved reader, you can guess I do not consider myself very familiar with functional programming, and while I followed and enjoyed most of this book, I had a very hard time here and there understanding what the author was trying to compress in as few tokens as possible.

[6]Both authors cite here, as well as others listed in the references come from the Netherlands, and I don't see this as a coincidence. There must be something special about the dutch language that attracts so many dutch and dutch-speaking scientists to the area of language recognition.

> Adding this intelligence to a single-threaded, possibly backtracking, parser is not always simple. Managing this intelligence for a parallel parser can become dauting, and even unfeasible.

The fact that the examples listed in [49] are all related to `COBOL` does not come as a surprise: `COBOL` is pure. Because of the intricacies of some of its constructs, and even more, by its mere size, `COBOL` is hard to parse and commonly used as a typicaly example of an abnormally difficult language to parse. In practice, however, it requires nothing more than a decent lookahead or equivalent facility – parallel (GLR), backtracking (`DURA`, PRECC or BTYACC) – and a working grammar[7].

On the other hand, when considering `FORTRAN` (and `PL/1`, even if workarounds can be used), the classical model depicted in figure 2.2 in page 21 *and on which all general parsing algorithms are based* does not apply. As we've seen earlier in this paper, parsing `C++` raises other, even more difficult issues. General parsing methods can be used, but they clearly are ill-suited to the task.

René Leermakers states as a surprise, almost a joke, that some of the people in the programming language parsing area don't understand why tools such as `yacc` and `lex` are not used to parse natural languages. After reading his work, we can understand why. On the other hand, I also believe that parsing programming languages, specially poorly designed ones, requires ad hoc techniques. One size does not fit all.

[7] Getting hold of a usable grammar for `COBOL` is far from being as easy as for Pascal or Ada, see [159].

Conclusion

These conclusions continue the charming tradition of the talmud style, started in [32], maintained in [31], and revived more recently in [160]. It aims at clearly separating the true content, the message, the plain truth from the commentaries, the more personal issues, the subjective view on the topic. This separation is implemented by a specific typesetting style. Talmud – understand: commentaries – paragraphs are indented with a slanted typeface, as opposed to the plain text, not indented and with a plain times roman typeface.

cuius rei
demonstrationem
sane detexi hanc
marginis exiguitas
non caperet

The tools

Of course, the most positive conclusion one can make about the tools (namely, `DURA` and `lexyt`) that have been developed as part of this work lies in two words: they work. They have been used in large production systems, and have proven to be reliable, versatile, and, most importantly, immensely productive. It is perhaps more accurate to state that they have *become* more reliable and accurate over the months. Error reporting and detection of abnormal conditions were – as often the case with purely theoretical and academic developments – the weak points of the toolset in its early releases.

It was not planned that way. When starting this work, I never thought it would take me so long, which is fortunate, because I'm not sure I would have started if I knew. I never thought it would take such a place in my life. I never thought I'd dream of it. I never thought it would follow me in large parts of the world. I debugged parsers in Indonesia, wrote documentation in Thailand, checked specification in Malaysia, optimized parser generation in Singapore; I redesigned the error recovery scheme when in Great-Britain, I reviewed drafts in South Africa and in China, I flew throughout the United States writing the last, and most painful chapters of this work. I optimized memory occupation in Switzerland, gathered statistics in trains all over Europe, and I read, my own prose as well as other authors, over and over again. I think I'll remember these three years as filled with a combination of constant travelling and constant thinking, reading, and working on this piece of work. Three very happy years indeed.

... a demonstra tionem sane detexi hanc marginis exiguitas non caperet.

As the section on related works shows extensively, there are myriads of parser generation tools out there. However, I believe that the tight integration of the parsing technology in the host language – YAFL – as provided by `DURA-lexyt` is rather unique, and is the most significant contribution of this work.

It is also significant that several features – such as the ability for a lexeme to backtrack from a significant token to a skippable token (See page 225) – have been added to enable the production of parsers for some extreme languages, and that there was no way I could have guessed that such features might even be useful before bumping into the real world problem that required them.

Regrets

Regrets, I had a few, but then again, too few to mention...

This is of course a tricky question: knowing what I know now, what would I do differently if I had to redevelop `DURA` and `lexyt`? Of course, my first answer would be simple: if I had to do it again, I simply wouldn't.

Slightly more seriously, there are a few things that would differ if we ever get to something such as `DURA`++ or Visual `lexyt`:

Cuius rei demonstrationem sane detexi. Hanc marginis exiguitas non caperet.

- I am too much of a coder. I enjoy putting well-written code together. Even if I fully support the statement about the importance of code in the introduction – as it was written three years ago – my only regret would be not to have paid enough attention to a more relevant division in classes and modules. I believe I should have analyzed *responsibilities* of the various classes in the system more thoroughly, ensuring that every piece of logic belongs to the most relevant class.

- After having produced this huge amount of code in literate programming form, I know for sure that if I had to do it again, I would have developed a tool to produce most of the cross referencing information automatically, after having performed thorough semantic analysis on the YAFL code. This tool would have cost me a couple of weeks of work, but would have spared me far more than that in boring and tedious typesetting effort. Literate programming is great, but when it comes to the documentation of large amounts of source code, a literate programming environment as a typesetting tool is not enough any longer. One needs more.

Literate programming

Of course, the first feeling when reaching this stage in such a huge document is relief. The only problem is that one cannot believe that a 1000 pages document is perfect by any standard unless totally out of one's mind. Having reached this concluding section, I know that there are a few zillions places in this document where an additional sentence, a reference,

> *an example or a figure might make the text more readable and more understandable. I am scared to death to the idea of a reader finding a true flaw – I should say, one of the true flaws I'm sure this text still contains – but I've had enough. I'll gladly face the lions. Terrified I am, but relieved I am too.*

Cuius rei demonstrationem sane detexi hanc marginis exiguitas non caperet

In the introduction, I claimed that literate programming was frustrating and rewarding. Experience have proven me right on both accounts. It involves a huge amount of intrinsically tedious work.

Maintaining `DURA` and `lexyt` based on this literate programming source code has been reasonably easy. The problem with such statements is their intrinsically limited scope. To measure the degree of improvement over a classical approach where code and related documentation, if any, are managed separately, one would have had to have the system developed twice by comparably skilled people, in comparable environments. This is of course totally unrealistic. So, beloved reader, you will have to take my word for it: it does help considerably. However, in the light of the amount of extra work it requires – a factor 3 is commonly accepted when comparing a "normal" software development effort with its literate programming counterpart – it can only be applied for mission critical code, or for publication purposes. Both of these criteria apply to this work, as it is being published, and as it has become a central component in the `RainCode` architecture.

It is also funny, after having investing so much time and effort in making the source as pleasant and readable as I could, how most readers basically skip that part as being ultimately uninteresting. A major point I was trying to make apparently did not appeal to the readership of this work: tools such as `DURA` and `lexyt` do not exist by nice theory only. They require craftmanship, and lots of mundane issues have to be taken care of. Some of the best ideas in this work lie in local implementation details, while the underlying technology is often simplistic. The volume of code, compared to the attached literature is yet another example that programming is an art [88]: it is made of 1% inspiration and 99% perspiration.

There is another, less expected side effect of literate programming which is barely mentioned in the literature: the obligation of explaining the internals of what you develop forces you to always prefer the simple way over the clever one, not so much because of how hard it is to develop, but because of how hard it gets to *explain*. Some techniques I avoided, not so much because I did not master them, but because I am not enough of a technical writer to be comfortable with the idea of explaining them.

Formal methods

The most significant disappointment of this work lies in the true usability of formal methods. As stated before, and as you might have noticed by having read even fragments of this document, I am more of an implementation person, even if with a formal bias. To me, opting for a formal specification was a way of putting up a bridge toward this exotic place, the world of formal methods.

I tried to be a well-behaved and environment-conscious tourist. I followed the advice of formal method people very thoroughly. I tried to integrate their culture, to understand and assimilate their way of thinking. I followed their advices thoroughly and consistenly: I never took a piece of specification for granted, I challenged every single glyph until feeling perfectly comfortable with it. I always tried to provide an abstract view on the problem at hand, rather than to mimic the implementation. For instance, Z sets have been preferred over sequences in the specification in all occurrences where the *order* of a collection was irrelevant, even if they are ultimately implemented as lists or arrays anyway.

Even if I would not call myself a formal method expert, I think I am as close to this world as a production-oriented individual will ever be. By investing only a fraction of the time I've invested in mastering the formal method approach in developing OO skils, any old-fashioned procedural programmer can take advantage of most of what OO offers.

> *I actually liked using formal methods. It is rather fun to build an intellectual image of a system. Defining a system's behaviour mathematically gives the feeling that all possible misunderstandings have been avoided, and it is a very rewarding feeling indeed. The trouble is, it is also a very mistaken feeling. After all, given a reasonably well-defined language semantics, the code is also a very precise description of a system's behaviour, and, if measured by how many people actually enjoy reading code, it is probably not very understandable. It is not sufficient to be precise: one must also build the reader intuition, and in my opinion, formal methods don't do a very good job in this area.*

Despite this considerable effort, I did not measure much of a return on investment for having used ObjectZ. I did not need formal specification for my own understanding. I am an implementation person, and I can deal with the large and complex implementation scaffoldings I develop with little effort. As far as I can say after having submitted draft versions of this document to various – all highly educated – individuals, it did not help them much in understanding what I was up to. Formal specification describe a system's state and behaviour precisely, but the result, while intelligible from a strictly mathematical point of view, does not help building up intuition. I don't think it has much to do with ObjectZ as a language: as you can see by going through the specifications listed in this work, I carefully avoided most of its syntactical idiosyncracies, and thoroughly introduced any of its feature which was not part of any reasonably mathematically educated reader's background.

Out of my experience using them, I see several problems with formal methods:

- The lack of a definition of the ultimate aim of formal methods. In other words, the absence of a clear message about what can be expected by using them. Who reads them? What is the profile of a reader who could use them proficiently? The implementer? The tester? A domain specialist? Someone must be able to gain something by reading the specification. By not clearly saying when they should be used, and what should be expected by using them, the formal method community discards any true usability evaluation effort.

- Formal methods also suffer from a somehow ambiguous attitude: must the specification be as abstract as possible, or must it somehow reflect the implementation? If it is too abstract, too disconnected from the implementation, it can be made understandable, but the reader accepts the author's claim about its correctness for granted. In other words, the reader has a very hard time trying to ensure that this abstract specification actually matches the implementation. The specification is then a description of the author's *intention*, with little if any guarantee that the resulting system matches this intention in any measurable way. On the other hand, specification languages can be used to describe the implementation accurately, up to a point where the reader does not see the abstract construction behind it.

This problem has been identified and addressed by integrated formal development methods such as B [161], where the specification undergoes a number of formal transformations, up to a point where it is directly mappable onto code. However, considering some of the trivial examples provided in the B literature, one can doubt it can be used proficiently for large and at least moderately complex systems such as this work. Among other limitations, B does not seem to manage dynamic systems gracefully. B clearly aims at the development of simple, but highly critical systems, where the problem at hand is not specially complex to master, but where errors must be avoided at any cost. It is no wonder several of the industrial references of B are related to embedded systems for the transportation industry.

- An expressivity problem. First order logic – as supporter by Z and ObjectZ – is intellectually tempting. It provides a clean and well defined semantic. It works fine with all the examples provided in the literature. On the other hand, describing a reasonably complex real-world system using such basic primitives is cumbersome, to say the least. For instance, in the ObjectZ specification included in this document, I had to define and make extensive use of references (ρ)(See page 979) to describe the actual behaviour of the system, including *shared objects*.

It is probably not excessive speculation to predict that the promoters of formal methods will claim that the problem lies with me – which, as a basic assumption, is perfectly possible. I think I am not mistaken when believeing that at some stage, I'll hear something about me having to spend more time learning how to use formal methods, how I should have spent more time in writing the specs, how I am probably presomptuous claiming that formal methods don't work while it is merely a question of me not having tried hard enough.

> *I cannot believe Fermat realized he started a 200 years tradition when scribbling notes in the margin of one of his books [162]. So beloved reader, if you feel like writing something amazing that will thrill generations of scientists, be my guest. There are over one thousand margins in this work. More than enough for a verbatim transcription of Gödel's theorem. Life is great.*

I can accept this argument, and perhaps, I'll even go back to work. I'll try to further improve the spec, admitting that my judgement was perhaps a bit hasty. I'll do that

Cuius rei
demonstrationem
sane detexi hanc
marginis exiguitas
non caperet

if the formal method community provides me with a deadline, a criterion, a measure, something that will allow me to decide at some stage the formal method *might* fail. There is no scientific truth without a point of *possible refutation* as defined by Sokal and Bricmont in [163], that is, the clear definition of what one is ready to accept as a proof or even just a sign that the thesis is untrue.

Of course, if the formal method community is right believing the fault lies at my side, I'll never reach that stage. If formal methods are as successful as they claim they are, I'll measure the advantage gained by using them long before having reached this failure point. However, in absence of such a failure point, there is no possible contradiction any longer. Whatever I might say, it can be claimed that formal methods did not work for me because I did not use them long enough. This is not science any longer. This is dogma.

> *Relief is of course not the only feeling. This work filled my days and my nights as no previous undertaking ever did. Emptyness is perhaps the most accurate description of how I feel now. There is a bit of sadness as well. "En dan?" What's next? I don't know. I don't even feel like thinking about what is going to come next. On the other hand, I would not want to leave you on a sad note. There is far more to rejoice than to complain about. All in all, beloved reader, thank you for having shared this with me. I've had a delightful time in your company. I sure hope you had the same.*

Chapter 10

Appendices

10.1 Changes to the YAFL environment

The YAFL environment has been used successfully for several years now, and the tools has shown to be stable, versatile and reliable. In most cases, it is now used as any other third generation programming language, and the fact that we developed it ourselves has little impact on the way we use it daily.

However, in the course of this work, we faced a few cases where some small improvements in some of the YAFL tools could mean a significant improvement for the project. This section summarizes these changes and improvement, and attempts to justify why they are useful for these project, why they are harmless or even better, useful for our existing base of YAFL systems as well as new systems that will be developed in the future.

10.2 Changes to the YAFL compiler

10.2.1 Dead code elimination

Dead code elimination is a technique that detects pieces of code that never will be executed due to a statically enforceable condition. For instance, figure 10.1 shows an example of code where a reasonably smart compiler can detect that one of the branches of the test will be executed on every occurence, remove the test, as well as the other branch.

Figure 10.1 Dead code elimination

```
IF ComplexCondition AND (1 = 2) THEN
  -- Do this
 ELSE
  -- Do that
  END;
```

Until we started developing `DURA` and `lexyt` , all the YAFL code we dealt with was hand-written. Admittedly, there is little chance that such bizarre code can be written by a decent programmer, and we never felt the need to include this – simple – feature in the YAFL compiler. However, things are different when one deals with code that is generated automatically.

Instead of requiring excessively intelligent tools that only generate minimal YAFL code, or even, relying on the hypothetical ability of the underlying C compiler to perform dead code elimination, we rather added that intelligence in the standard YAFL compiler.

Quite predictably, this optimization had no effect on our existing code base.

10.2.2 Optimized code generation for trivial `CASE` statements

Similarly to dead code elimination, when dealing with automatically-generated code, one sometimes encounters strange pieces of code such as depicted in figure 10.2.

Figure 10.2 Degenerated `CASE` statement

```
CASE i OF
  1:
    -- Do this
    END;
 ELSE
  -- Do that
  END;
```

The original YAFL compiler did generate “standard” code for this statement (see figure 10.3), even if a far more efficient and compact scheme could have been used instead.

Figure 10.3 `C` code produce by a degenerated `CASE` statement

```
{
  int Y_081 = Y_i;
  switch (Y_081)
    {
      case 1:
        /* Do this */
        break;
      default:
        /* Do that */
    }
}
```

The YAFL compiler now generates an optimized `if` construct for such degenerated `CASE` statements. It might have been worth the trouble to first investigate what kind of optimization the underlying `C` compiler might have produced on such degenerated code, but it seemed safer not to depend on the availability of such a feature that might or might not have been supported.

10.2.3 Array fill optimization

The YAFL profiling system counts *statements* rather than absolute execution times. This simplistic approach biases measurements, but detects major bottlenecks, and offers the additional advantage of being utterly portable, since it does not require any intervention below the level of the generated `C` code.

When profiling early versions of `lexyt` and `DURA`, it appeared that *at least, in terms of statement counts*, a significant amount of time was spent in reinitializing arrays with 0 (or VOID).

We could have cheated our profiling reports by replacing these filling loops (with a statement count equal to the number of elements in the array) by a new allocation (with a statement count of one), but this would have been a poor engineering decision, since it would have made the system actually *slower* for the sake of partially reliable performance analysis.

Hence, we added pattern matching code in the YAFL compiler to detect several easily optimizable constructs:

- Filling an array slice with a constant value, filling the entire array being considered as a special case of this.
- Filling an array slice with a constant value which is equal to zero (“`0`”, “`0.0`”, `FALSE`, `VOID`) in which case a memory fill function can be used
- Copying an array slice onto another array

It is worth mentioning that even in cases where apparently, there is little to gain, the fact that one can avoid the repeated cost of YAFL array accesses can make a very significant difference (see figure 10.4).

10.2.4 Externalized code generation

When the YAFL project started, we considered the *compiler support* feature as a nice addition to an otherwise quite conventional third generation programming language. It is the ability to reuse whole parts of the compiler through inheritance. An entire mechanic implements this feature, based on special classes called *creators* which, according to Gamma's terminology[84], are *factories* or *abstract factories*.

Early designs of the YAFL compiler were based on a simple architecture where classes roughly implemented the non-terminals of the grammar definition, and where all facilities attached to these non-terminals (type checking, code generation) were attached to these classes as methods.

This scheme was satisfactory, until more and more of the developments that were done on extending the standard YAFL compiler had some form of relationship with code generation ([164], [23], [57] and [20] as well as this work) up to a point where it clearly would have been more convenient to decouple code generation as a service from the rest of the compiler. This has been achieved by the *externalization of the code generation*, which is a rather systematic process and which is described extensively in [31].

The rather strong coupling between a non-terminal class and its attached code generation class (the former can only be generated by the latter, the latter can only generate code for the former) is formalized by a bi-directional parameterization of the classes. For instance, we have:

```
CLASS LoopStatement;
  INHERITS CompoundStatement(LoopStatCodeGenerator);
...
```

and the corresponding

```
CLASS LoopStatCodeGenerator(CtxType IN GcContext);
  INHERITS StatementCodeGenerator(LoopStatement, CtxType);
...
```

In practice, this means that one can now write new code generators or extend existing ones without altering the existing compiler at all. This also means that such code generators can be *linked*, that is, that a compilation run is attached to several (and possibly, independent) code generation passes that are invoked one after the other automatically. For instance, the plain YAFL compileras invokes two entirely separate and independent code generators to produce intermediate `C` code the metrics information.

In the context of this thesis, externalized code generation has been used to add the generation of `DURA` intermediate files to the common code generation passes of the compiler.

Figure 10.4 Comparing naive and optimized code generation

Given the piece of code:

```
VAR
  p: ARRAY OF INTEGER;
  i: INTEGER;
BEGIN
...
FOR i := 0 TO p.SIZE - 1 DO
  p[i] := 1876;
  END;
```

a naive translation would yield:

```
obj_ptr Y_p;
int Y_i;
...
for (Y_i=0; Y_i<ARRAY_SIZE(p); Y_i++)
  {
    ((int *)Y_p) [i] = 1876;
  }
```

while the optimized scheme based on patterm matching, would yield:

```
obj_ptr Y_p;
int Y_i;
...
{
  int * Y_00 = Y_p;
  for (Y_i=0; Y_i<ARRAY_SIZE(p); Y_i++)
    *(Y_00++) = 1876;
}
```

and, if we assigned “`0`” instead of “`1876`”:

```
obj_ptr Y_p;
int Y_i;
...
memset(Y_p,0,ARRAY_SIZE(p)*sizeof(int));
```

10.3 Literate programming

This project is the first production-level attempt to use YAFL together with literate programming techniques. In theory, literate programming should have little if any impact on the underlying tools, but, in practice, it is an entirely different matter.

10.3.1 Compiler error-reporting

The first topic that must be considered is compiler error reporting. The source code that the compiler receives has been generated automatically from the `noweb` source. Hence, the line numbers that are reported in conjunction with errors do not refer to the true source code, that is, the `noweb` source code.

When using `notangle` to generate the intermediate YAFL source code, coded comments are inserted, for instance:

```
--<ylex.nw>14
```

to indicate that this intermediate code position corresponds to the line number 14 in the original source file "`ylex.nw`". Functionally, this feature is similar to `C`'s `#line` directive. However, due to `C`'s inclusion of a line-based preprocessor, such a directive is smoothly integrated in the rest of the language. YAFL is not line-oriented at all. Hence, while one might be tempted to integrate such a source-line resynchronization feature in the language definition, it soon proves to be awkward.

Instead of making the *language* `noweb`-aware, we made the compiler `noweb`-aware. An error processing option `-el` has been added. If this option is enabled, when displaying an error message, the intermediate YAFL source is read, recomputes the true source code position based on the coded comments as shown above, and issues a comprehensive error message, indicating the position in the original literate programming file.

In order to avoid that this entire processing is repeated for every error, the table of correspondance between the original source positions and the intermediate source code positions is cached, so an arbitrary number of errors that have been detected within a single file only require that this table is built once.

10.3.2 Pitfall

The scheme describe above suffers from a relatively minor pitfall. In plain YAFL, when one only alters an implementation module, client (understand: external) modules don't have to be recompiled. In a literate programming system, since both the definition and implementation modules are described withing a single "*.nw*" file, when one inserts or deletes lines, the numbered comments that are inserted into the generated YAFL source code might change, indirectly updating the definition modules, even if the original modification is only related to the implementation part of the module.

In practice, this limitation had no undesired effect beyond the sometimes unnecessary recompilations of modules which have not changed except for comments.

10.4 ObjectZ useful definitions

This section provides several useful ObjectZ definitions that are used throughout the ObjectZ schemas and classes in this document. They are not specific to grammatical specifications and could been reused in other contexts.

10.4.1 Simple type definitions

Well, at the beginning, we have characters.

$$[CHAR]$$

then, character strings are a natural extension:

$$STRING == \text{seq}\, CHAR$$

We define useful integer subsets:

$$\begin{array}{|l} \mathbb{N}_8 = \{x \in \mathbb{N} \mid x < 2^8\} \\ \mathbb{N}_{16} = \{x \in \mathbb{N} \mid x < 2^{16}\} \\ \mathbb{N}_{32} = \{x \in \mathbb{N} \mid x < 2^{32}\} \end{array}$$

and commonly used functions:

$$\begin{array}{|l} high_{16} : \mathbb{N}_{16} \mapsto \mathbb{N}_8 \\ low_{16} : \mathbb{N}_{16} \mapsto \mathbb{N}_8 \\ \hline \forall\, x \in \mathbb{N}_{16} \bullet x = high_{16}(x) \times 2^8 + low_{16}(x) \end{array}$$

$$\begin{array}{|l} high_{32} : \mathbb{N}_{32} \mapsto \mathbb{N}_{16} \\ low_{32} : \mathbb{N}_{32} \mapsto \mathbb{N}_{16} \\ \hline \forall\, x \in \mathbb{N}_{32} \bullet x = high_{32}(x) \times 2^{16} + low_{32}(x) \end{array}$$

10.4.2 Total and partial orders, generically and on sequences

The following generic schema is a slightly altered version of a more permissive one, present in the Z toolkit and which is described in [149]. The original version of *PartialOrder* defined a reflexive relation ($\leq$) while the one below describes an anti-reflexive one.

$$\begin{array}{l} PartialOrder[X] == \\ \quad \{\_ < \_ : X \leftrightarrow X \mid (\forall\, x, y, z : X \bullet \\ \qquad x \not< x \land \\ \qquad\quad (x < y \Rightarrow \neg(y < x)) \land \\ \qquad\qquad (x < y \land y < z) \Rightarrow x < z)\} \end{array}$$

Again, based on [149], the following extension proved useful:

$$
\begin{array}{l}
TotalOrder[X] == \\
\quad \{\_ < \_ : PartialOrder[X] \mid \\
\qquad (\forall x, y : X \bullet (x < y) \lor (y < x) \lor (x = y))\}
\end{array}
$$

$Sorted$ denotes all sorted sequences of the generic class X.

$$
\begin{array}{|l}
[X] \\
\hline
Sorted : \mathbb{P}\,\mathrm{seq}\ X \\
\hline
l \in Sorted \Leftrightarrow \forall i, j : 1..\#l \bullet \\
\qquad i < j \Rightarrow l(i) \le l(j)
\end{array}
$$

The following schema explains how generic sequences can be compared lexicographically based on the comparison function "$<$" applied on their elements.

$$
\begin{array}{|l}
CompareSequences\,[X] \\
\hline
\_ < \_ : \mathrm{seq}\ X \leftrightarrow \mathrm{seq}\ X \\
\hline
\forall a \in \mathrm{seq}\ X \bullet \neg(a < a)
\end{array}
$$

seq_1 denotes a non-empty sequence, as opposed to a plain sequence seq that can be empty: $\langle\rangle$.

$$
\begin{array}{|l}
\forall a \in \mathrm{seq}_1\ X \bullet \langle\rangle < a \\
\forall a, b \in \mathrm{seq}_1\ X \mid head\ a < head\ b \bullet a < b \\
\forall a, b \in \mathrm{seq}_1\ X \mid head\ a = head\ b \bullet \\
\qquad a < b \Leftrightarrow tail\ a < tail\ b
\end{array}
$$

10.4.3 Extracting a sequence slice

$Slice$ defines the extraction of a sequence slice, similarly to YAFL's predefined method SLICE.

$$
\begin{array}{|l}
[X] \\
\hline
Slice : \mathrm{seq}\ X \times \mathbb{N} \times \mathbb{N} \rightarrow \mathrm{seq}\ X \\
TopSlice : \mathrm{seq}\ X \times \mathbb{N} \rightarrow \mathrm{seq}\ X \\
BottomSlice : \mathrm{seq}\ X \times \mathbb{N} \rightarrow \mathrm{seq}\ X \\
\hline
Slice(l, 1, len) = l \Leftrightarrow len = \#l \\
Slice(l, 1, len) = Slice(front\ l, 1, len) \Leftrightarrow len < \#l \\
Slice(l, p, len) = Slice(tail\ l, p - 1, len) \Leftrightarrow p > 1 \\
\\
TopSlice(l, len) = l \Leftrightarrow len = \#l \\
TopSlice(l, len) = TopSlice(front\ l, len) \Leftrightarrow len < \#l \\
\\
BottomSlice(l, len) = l \Leftrightarrow len = \#l \\
BottomSlice(l, len) = BottomSlice(tail\ l, len) \Leftrightarrow len < \#l
\end{array}
$$

10.4.4 Specifying references

Z and ObjectZ have a strictly value-based semantics. There is no such thing as references to objects in such a way that an object can be altered through one reference while these alterations can be accessed through other references.

Since one of the major assets of object-orientation lies in its ability to build complex webs of objects which communicate with each other using a reference-based semantics, we need some way of expressing such references.

This is the purpose of the following generic schema: for every type $Type$, it defines *another type* named $\rho\,Type$, which is a reference to a $Type$. This schema also defines a dereferencing operation noted δ which returns the value referred to by a reference.

$[Type]$

$[\rho\,Type]$

$\delta : \rho\,Type \mapsto Type$

δ^1 is the generalization of δ to a set of references.

$\delta^1 : \mathbb{P}\,\rho\,Type \mapsto \mathbb{P}\,Type$

Λ is defined as the null pointer for all reference types.

$\Lambda \in \rho\,Type$

$\delta^1(s) = \bigcup_{e \in s}\{\delta\, e\}$

10.4.5 Intervals

$[Type]$

$Interval\ Type == \mathbb{P}\ Type$

Given a lower and an upper bound, $MakeInterval$ returns a new $Interval$.

$MakeInterval : Type \times Type \mapsto Interval\ Type$

The following constraint expresses the fact that an interval is a contiguous subset of the reference type. It assumes the existence of a total $\leq$ relationship, which is by the way implied by the min and max functions which extract resp. the minimum and the maximum of the elements of the set they are applied to.

$\forall\, Int : Interval \bullet$
$\quad \forall\, i \geq min\ Int \wedge i \leq max\ Int \bullet$
$\qquad i \in Int$

$MakeInterval(l, r) = \{x \in Type \mid l \leq x \leq r\}$

Bibliography

[1] Adobe Systems Inc., editor. *PostScript Language Reference Manual*. Addison-Wesley, 1990. ISBN 0-20-0118127-4.

[2] HTML 4.0 specification. URL=http://www.w3.org/TR/REC-html40/, 1997.

[3] W3C. Extensible markup language (xml) 1.0. URL=http://www.w3.org/TR/REC-xml.

[4] Ian S. Graham, Liam Quin. *XML Specification Guide*. John Wiley & Sonc Inc., 605 Third Avenue, New York, NY 10158-0012, 1999. ISBN 0-471-32753-0.

[5] Michael R. Garey, David S. Johnson. *Computers and Intractability: A Guide to the Theory of Np-Completeness*. W.H. Freeman, 1979. ISBN 0-71-67104-55.

[6] Oren Patashnik. BIBTEXing. Documentation for general BIBTEXusers. Electronic document coming with the BIBTEXdistribution.

[7] Oren Patashnik. Designing BIBTEX styles. The part of BIBTEXthat's not meant for general users. Electronic document coming with the BIBTEX distribution.

[8] Françoise Dolto. *Tout est langage*. LGF, 1989.

[9] K. Iverson. *A Programming Language*. John Wiley & Sonc Inc., New-York, 1962.

[10] M. F. Cowlishaw. *The Rexx Language*. Prentice Hall, Englewood Cliffs, 1985.

[11] Tim Alton. *Programming With Python*. Prima Publishing, 1999. ISBN 0-76-152334-0.

[12] Python Language Website. URL=http://www.python.org/, 1999.

[13] Niklaus Wirth and Jurg Gutknecht. *Project OBERON, The Design of an Operating System and Compiler*. Addison-Wesley, New York, NY, 1992.

[14] Chris Fraser David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995. ISBN 0-8053-1670-1.

[15] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992. Distributed by the University of Chicago Press.

[16] Andrew S. Tanenbaum. *Operating Systems, Design and Implementation*. Prentice-Hall International Editions, 1987. ISBN 0-13-637331-3.

[17] Jean-Christophe Real. rypp: Analyse Orientée-objet du langage PL/1. Master's thesis, Université Libre de Bruxelles, 1994.

[18] International Business Machines Corp. *OS and DOS PL/1 Language Reference Manual*, 1981.

[19] Terence J. Parr. LL and LR Translator Need K. *ACM SIGPLAN Notices*, 31(2), February 1996.

[20] Lionel Ferette. `cool` : a compiler as a compiler-compiler. Master's thesis, Université Libre de Bruxelles, 1996.

[21] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[22] Jean Muepu. YAFL/WEB: la programmation littéraire. Master's thesis, Université Libre de Bruxelles, 1994.

[23] Laurent Ferier. Global Optimization in YAFL . Master's thesis, Université Libre de Bruxelles, 1996.

[24] Ullman Aho, Sethi. *Compilers: principles, techniques and tools ( )*. Addison-Wesley, 1986. ISBN 0-201-10194-7.

[25] Josef Grosch. The CoCoLab COBOL parser. URL=http://www.cocolab.de/html/cobol.htm.

[26] J. M. Spivey. An introduction to Z and formal specifications. *IEE/BCS Software Engineering Journal*, 4(1):40–50, January 1989.

[27] Roger Duke, Paul King, Gordon Rose, and Graeme Smith. The object-Z specification language - version 1. Technical Report 91-1, Software Verification Research Centre; The University of Queensland, Queensland 4072, May 1991.

[28] The LEGO Group. Official lego world wide web site. URL=http://www.lego.com.

[29] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesley, 1979. ISBN 0-201-02988-X.

[30] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13/2, 1970.

[31] Darius Blasband. *The YAFL Programming Language, 2nd edition*. PhiDaNi Software, 1994.

[32] Darius Blasband. The YAFL Programming Language. Master's thesis, Université Libre de Bruxelles, 1991.

[33] Norman Ramsey. The `noweb` literate programming tool home page. URL=http://www.cs.virginia.edu/~nr/noweb/.

[34] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994.

[35] Data General Corporation - AT&T. *Using the Documenter's Tool Kit on the DG/UX System*, 1988.

[36] Klaas Sikkel. *Parsing schemata: a framework for specification and analysis of parsing algorithms*. Springer-Verlag, 1997. ISBN 3-540-61650-0.

[37] Douglas R. Hofstader. *Godel, Escher, Bach: An Eternal Golden Braid*. 1979. ISBN 0-46-502656-7.

[38] W. Pitts W.S. McCulloch. A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophysics*, 5:115–133, 1943.

[39] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[40] M. E. Lesk. Lex — a lexical analyzer generator. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, N.J., 1975.

[41] Jean-Eric Waroquier. Personal communication, 1996.

[42] William M. Fuori. *Introduction to American National Standard COBOL*. McGraw-Hill, New York, 1975.

[43] MicroFocus COBOL. URL=http://www.merant.com/ads/products/.

[44] The compilers usenet newsgroup. URL=news://usenet.comp.compilers.

[45] Allen I. Holub. *Compiler design in C*. Prentice Hall International Editions, 1990.

[46] Raymond Devillers. Personal communication, 1999.

[47] Vern Paxson. Flex: a fast scanner generator. URL=ftp://ftp.uu.net/packages/gnu/flex/flex-2.5.4a.tar.gz.

[48] Jon Bentley. *More Programming Pearls, Confessions of a Coder*. Addison-Wesley, 1990. ISBN 0-201-11889-0.

[49] M.G.J. van den Brand, A. Sellink and C. Verhoef. Current Parsing Techniques in Software Renovation Considered Harmful. *International Workshop on Program Comprehension, Ischia, Italy*, june 1998.

[50] W.F. Clocksin and C.S.Mellish. *Programming in Prolog*. Springer-Verlag, fourth edition, 1994. ISBN 3-540-58350-5.

[51] Niklaus Wirth. The design of a Pascal compiler. *Software – Practice and Experience*, 1:4:309–333, 1971.

[52] Niklaus Wirth. *Programming in Modula II, 4th edition*. Springer-Verlag, 1988. ISBN 3-540-50150-9.

[53] Martin. Reiser. *The Oberon System User Guide and Programmer's Manual*. ACM Press, New York, NY, 1991.

[54] Waite W.M. The Cost of Lexical Analysis. *Software Practice and Experience*, 16(5), may 1986.

[55] Gray, R.W. γ-GLA - A generator for lexical analyzers that programmers can use. In *USENIX Conference Proceedings*, pages 147–160, 1988.

[56] Donald D. Cowan Peter Bumbulis. Re2c: A more versatile scanner generator. *LOPLAS 2(1-4)*, pages 70–84, 1993.

[57] Didier Barzin. GNU-YAFL. Master's thesis, Université Libre de Bruxelles, 1998.

[58] Frederick P. Brooks, Jr. *The mythical man-month*. Addison-Wesley, 1975. ISBN 0-201-00650-2.

[59] Donald E. Knuth. On the translation of languages from left to right. *Information and control*, 8:607–639, 1965.

[60] Vadim Maslov, Chris Dodd. BTYacc – Backtracking `yacc` – home page. URL=http://www.siber.com/btyacc/.

[61] Peter Breuer Jonathan Bowen. A PREttier Compiler-Compiler: Generating Higher Order Parsers in C. *Software Practice and Experience*, 25(1):1263–1297, nov 1995.

[62] Peter Breuer Jonathan Bowen. Occam' Razor: The Cutting Edge on Parser Technology. In *TOULOUSE 92: Fifth International Conference on Software Engineering and its Applications*, dec 1992.

[63] Josef Grosch. Lark - An LR(1) Parser Generator With Backtracking. Technical report, CoCoLab - Datenverarbeitung, april 1998.

[64] Terence J. Parr and Russell W. Quong. ANTLR: A Predicated- LL(k) Parser Generator. *Software - Practice and Experience*, 25(7):789–810, july 1995.

[65] The Java Compiler Compiler. URL=http://www.suntest.com/JavaCC.

[66] Masaru Tomita. *Efficient parsing for natural language*. Kluwer Academic Publishers, 1986.

[67] Alain Pirotte. *"It is possible to build a family of parsers for general context-free grammars; which have the same theoretical bounds for time and space performance as Earley's algorithms but which are more efficient in practice for most grammars"*. PhD thesis, Université Libre de Bruxelles, 1976.

[68] M. Snelling M. Bouckaert, A. Pirotte. Efficient parsing algorithms for general context-free grammars. *Information Sciences*, 8(1):1–26, 1975.

[69] Jay Earley. *An efficient context-free parsing algorithm*. PhD thesis, Carnegie-Mellon University, 1968.

[70] Ceriel J.H. Jacobs Dick Grune. *Parsing Techniques - A Practical Guide*. Ellis Horwood, Chichester, England, 1990. ISBN 0-13-651431-6.

[71] R.W. Floyd. Syntactic analysis and operator precedence. *Communications of the ACM*, 5(10):316–333, 1962.

[72] Frank DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, July 1971.

[73] Thomas J. Pennello Frank DeRemer. Efficient computation of LALR(1) lookahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, October 1982.

[74] Ole Lehrmann Madsen Bent Bruun Kristensen. Methods for computing LALR(k) lookahead. *ACM Transactions on Programming Languages and Systems*, 3(1):60–82, January 1981.

[75] R. Duke, P. King, G. A. Rose, and G. Smith. The Object-Z specification language: Version 1. Technical Report 91-1, Department of Computer Science, University of Queensland, St. Lucia 4072, Australia, April 1991.

[76] Nozohoor-Farshi R. Handling of Ill-designed Grammars in Tomita's Parsing Algorithm. In *International Workshop on Parsing Technologies (IWPT'89)*, pages 182–192. Carnegie Mellon University, Pittsburgh, Pa., 1989.

[77] René Leermakers. *The functional treatment of parsing*, volume 242. Kluwer Academic Publishers, 1993.

[78] Nederhof, M.J. Generalized Left-Corner Parsing. In *6th Meeting of the European Chapter of the Association of Computational Linguistic (EACL'93)*, pages 305–314. Utrecht, 1993.

[79] Nederhof M.J., Sarbo J.J. Increasing the Applicability of LR Parsing. In *3rd International Workshop on Parsing Technologies (IWPT'93)*, pages 187–201. Tilburg and Durbuy, Netherlands/Belgium, 1993.

[80] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[81] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Inc., 1.39 edition, January 1991.

[82] The GNAT Team. *The GNAT project: A Gnu-Ada9X compiler*. Courant Institute of Mathematical Society, New-York University, 1995.

[83] Darius Blasband and Jean-Christophe Real. All-purpose quantifiers in an OO language. *Proceedings of TOOLS Asia 1998*, 1998.

[84] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[85] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1988. ISBN 0-201-06673-4.

[86] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992. ISBN 0-13-247925-7.

[87] Philippe Elinck. De la programmation-conception orienté-objets. Master's thesis, Université Libre de Bruxelles, 1988.

[88] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. McGraw-Hill, 1998. ISBN 0-201-89684-2.

[89] Corey Yeatman Bruce J. McKenzie and Lorraine de Vere. Error repair in shift-reduce parsers. *ACM transaction on Programming Languages and Systems*, 17,4:672–689, 1995.

[90] Norman Ramsey. *Eliminating spurious error messages using exceptions, polymorphism, and higher-order functions*. Dept of Computer Science, University of Virginia, 1996.

[91] CERA Bank. CERA Bank World Wide Web Site. URL=http://www.cera.be.

[92] Miguel de Cervantes Saavedra. *El ingenioso hidalgo don Quixotte de la Mancha*. Roger Velpius y Humberto Antonio, impressores de sus Altezas, en l'Aguila de oro, cerca de Palacio (I), Humberto Antonio, impressor jurado cerca del palacio (II), Bruselas, Belgica, 1611(I), 1616(II).

[93] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[94] J. M. Spivey. *The* fUZZ *Manual*. Computing Science Consultancy, 34 Westlands Grove, Stockton Lane, York YO3 0EF, UK, 2nd edition, July 1992.

[95] Xiaoping Jia. *ZTC: A Type Checker for Z – User's Guide*. Institute for Software Engineering, Department of Computer Science and Information Systems, DePaul University, Chicago, IL 60604, USA, 1994.

[96] M.A. Hewitt, C.M. O'Halloran and C.T. Sennett. Experiences with PiZA, an Animator for Z. In *ZUM' 97, the Z Formal Specification Notation*, volume 10, pages 37–45, 1997.

[97] M. Saaltink. Z and Eves. Workshops in Computing, pages 223–242. Springer-Verlag, 1992.

[98] University of Queensland Software Verification Research Centre. Wizard home page. URL=http://svrc.it.uq.edu.au/Object-Z/pages/Wizard.html.

[99] Josef Grosch. The Cocktail toolbox. URL=http://www.gmd.de/SCAI/lab/adaptor/cocktail.html, 1995.

[100] Peter Breuer Jonathan Bowen. PRECC support for parsing the Z notation. URL=http://www.cs.reading.ac.uk/archive/z/precc.

[101] RainCode. RainCode home page. URL=http://www.raincode.com.

[102] Siber Systems. Siber Systems home page, maker of `CobolTransformer`. URL=http://www.siber.com.

[103] Tom Christiansen Larry Wall and Randal L. Schwartz. *Programming Perl, 2nd Edition*. O'Reilly Associates, Inc., 1996.

[104] Lewis J. and Loftus W. *Java Software Solutions - Foundations of Program Design*. Addison Wesley, 1998. ISBN 0-201-12692-3.

[105] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, USA, 1990. ISBN 0-201-51459-1.

[106] Bernard Rasson. Conception d'un metteur au point pour le langage orienté-objet YAFL . Master's thesis, Université Libre de Bruxelles, 1993.

[107] Kirby Angell. Examining JPython. *Dr. Dobb's Journal*, 298:78–83, April 1999.

[108] The Python Software Activity. JPython home page. URL=http://www.jpython.org.

[109] Leslie Lamport. *LaTeX— A Document Preparation System — User's Guide and Reference Manual*. Addison-Wesley, Reading, 1985.

[110] *American National Standard X3.159-1989. The C Programming Language*. ANSI, New York, 1989.

[111] DoD. *Ada Reference Manual V6.0*, 1994.

[112] François Soumillion. RainCode for C++. Master's thesis, Université Libre de Bruxelles, 1999.

[113] Paul Klint, Eelco Visser. Using Filters for the Disambiguation of Context-free Grammars. Technical Report P9426, Programming Research Group, University of Amsterdam, december 1994.

[114] A. Van Wijngaarden et al. Revised Report on the Algorithmic Language ALGOL 68. *Acta Informatica*, 5:1–236, 1975.

[115] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, Inc., 1997. ISBN 1-55860-320-4.

[116] Chris Verhoef. Personal communication, 1999.

[117] M.G.J. van den Brand, P. Klint and C. Verhoef. Core technologies for system renovation. *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of LNCS:235–2575, 1996.

[118] Information technology – Database languages – SQL, 1992. Document Number: ISO/IEC 9075:1992.

[119] The Aflex and Ayacc home page. URL=http://www.ics.uci.edu/ arcadia/Aflex-Ayacc/aflex-ayacc.html, 1994.

[120] Mortice Kern Systems. MKS Lex and YACC. URL=http://www.mks.com/products/interop/ly/.

[121] Visual Parse++. URL=http://www.sand-stone.com/prod02.htm, 1999.

[122] C. Scott Ananian Scott Hudson, Frank Flannery. The JavaCUP home page. URL=http://www.cs.princeton.edu/%7Eappel/modern/java/CUP/, 1999.

[123] The Bison home page. URL=http://www.gnu.org/software/bison/bison.html.

[124] Thinkage YAY parser generator home page. URL=http://www.thinkage.on.ca/shareware/yay.

[125] Richard Hipp. The Lemon parser generator home page. URL=http://www.hwaci.com/sw/lemon/lemon.html.

[126] Hans J. Boehm, M. Weiser, A.J. Demers. A garbage collector for C and C++. URL=http://www.hpl.hp.com/personal/Hans_Boehm/gc/, 1999.

[127] Hans J. Boehm, M. Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, pages 807–820, September 1988.

[128] Zerksis D. Umrigar. The ZYacc home page. URL=http://opal.cs.binghamton.edu/ zdu/zyacc/.

[129] Delphi. URL=http://www.borland.com/delphi/, 1999.

[130] Hans Peter Mössenböck. A generator for fast compiler front-ends. Computing Science Technical Report No. 127, Institut für Computersystem, ETH Zürich, 1990.

[131] P. Rechenberg H. Mössenböck. *A Compiler-Generator for Microcomputers*. Prentice-Hall International Edititions, 1989.

[132] Niklaus Wirth. *Compilerbau*. Teubner Studienbücher, 1986.

[133] Etienne Gagnon. SableCC, and Object-Oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal, 1998.

[134] Etienne Gagnon. The SableCC home page. URL=http://www.sable.mcgill.ca/sablecc/.

[135] Josef Grosch. The Cocktail toolbox (Commercial version). URL=http://www.cocolab.de/html/cocktail.htm, 1999.

[136] Terence J. Parr. *Obtaining Practical Variants of LL(k) and LR(k) for k>1 by Splitting the Atomic k-tuple*. PhD thesis, Purdue University, West Lafayette, Indiana, august 1993.

[137] S.H. Unger. A global parser for context-free phrase structure grammars. *Communications of the ACM*, 11:240–247, April 1968.

[138] D.H. Younger. Recognition of context-free languages in time n^3. *Inform. Control*, 10(2):189–208, feb 1967.

[139] T. Kasami, K. Torii. A syntax-analysis procedure for unambiguous context-free grammars. *Journal of the ACM*, 16(3):423–431, july 1969.

[140] B. Sheil. Observations on context-free parsing. *Statistical Methods in Linguistics*, pages 71–109, 1976.

[141] Ceriel J.H. Jacobs Dick Grune. Parsing Techniques - A Practical Guide. URL=http://www.few.vu.nl/%7Edick/PTAPG.html, 1999.

[142] Jeffrey Kegler. Marpa extension to TCL to produce Earley parsers. URL=ftp://ftp.best.com:/pub/jeffrey/marpa/.

[143] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[144] Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. In *Proceedings of the Second Colloquium on Automata, Language and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer-Verlag, 1974.

[145] T.A. Wagner and S.L. Graham. Incremental analysis for real programming languages. *SIGPLAN notices*, 32(5):31–43, 1997. Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

[146] R. Nigel Horspool Philippe McLean. A Faster Earley Parser. In *Compiler Construction 1996*, volume 6, pages 281–293, 1996.

[147] R. Nigel Horspool John Aycock. Faster generalized LR Parsing. In *Compiler Construction 1999*, volume 8, pages 32–46, 1999.

[148] Peter Breuer Jonathan Bowen. A Concrete Grammar for Z. Technical Report PRG-TR-22-95, Oxford University Computing Laboratory, September 1995.

[149] Jonathan Bowen. *Formal specification and documentation using Z.* Thomson International Computer Press, Berkshire House, High Holborn, London WC1V7AA, United Kingdom, 1995.

[150] A. van Deursen. *Introducing ASF+SDF. In Executable Language Definitions*. PhD thesis, University of Amsterdam, 1994.

[151] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

[152] J. Heering, P.R.H. Hendriks, P. Klint and J. Rekers. The syntax definition formalism SDF – Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[153] R. Nigel Horspool. ILALR: An Incremental Generator of LALR(1) Parsers. In *Compiler Construction*, pages 128–136, 1988.

[154] R. Nigel Horspool. Incremental Generation of LR Parsers. In *Computer Languages*, volume 15(4), pages 205–223, 1990.

[155] D.J. Salomon and G.V. Cormack. Scannerless NSLR(1) parsing of programming languages. *SIGPLAN notices*, 24(7):170–178, 1989.

[156] D.J. Salomon and G.V. Cormack. The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Technical Report 95/06, Department of Computer Science, University of Manitoba, Winnipeg, Canade, 1995.

[157] Eelco Visser. Scannerless Generalized LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, august 1997.

[158] G.H. Roberts. An LR Analog to Recursive Descent. *ACM SIGPLAN Notices*, 23(8):23–29, August 1988.

[159] M.G.J. van den Brand, M.P.A. Sellink and C. Verhoef. Obtaining a COBOL grammar from legacy code for reengineering purposes. In *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*. Springer-Verlag, 1997.

[160] Arnaud Cormier. Questions: including declarative programming expressiveness in the Object-Oriented world. Master's thesis, Université Libre de Bruxelles, 1999.

[161] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Pr., 1996. ISBN 0-52-14961-95.

[162] Simon Singh. *Fermat's Enigma*. Anchor books, New-York, 1998. ISBN 0-385-49362-2.

[163] Alan D. Sokal, Jean Bricmont. *Impostures intellectuelles*. O. Jacob, 1997. ISBN 2-73-81050-33.

[164] Jerome Verstrynge. The YMELT Project. In *The First* $Y^{A}F_{L}$ *Conference - YEAH*, 1996.

Index

C