Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System

Martin JOURDAN, Didier PARIGOT, Catherine JULIÉ,
Olivier DURIN & Carole LE BELLEC
INRIA*

Abstract

FNC-2 is a new attribute grammar processing system aiming at expressive power, efficiency, ease of use and versatility. Its development at INRIA started in 1986, and a first running prototype is available since early 1989. Its most important features are: efficient exhaustive and incremental visit-sequence-based evaluation of strongly (absolutely) non-circular AGs; extensive space optimizations; a specially-designed AG-description language, with provisions for true modularity; portability and versatility of the generated evaluators; complete environment for application development. This paper briefly describes the design and implementation of FNC-2 and its peripherals. Then preliminary experience with the system is reported.

1 Introduction and Design Requirements

Since Knuth's seminal paper introducing attribute grammars (AGs) [34], it has been widely recognized that this method is quite attractive for specifying every kind of syntax-directed computation, the most obvious application being compiler construction. Apart from pure specification-level features—declarativeness, structure, locality of reference—, an important advantage of AGs is that they are executable, i.e., it is possible to automatically construct, from an AG specifying some computation, a program which implements it.

*Authors' address: INRIA, Domaine de Voluceau, Rocquencourt, BP 105, F-78153 LE CHESNAY Cedex, France. E-mail: {jourdan, parigot, julie, durin, lebellec}@minos.inria.fr. C. Julié is also with Laboratoire d'Informatique Fondamentale d'Orléans, Université d'Orléans, BP 6759, F-45067 ORLÉANS Cedex 2, France. E-mail: julie@univ-orleans.fr.

Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation. White Plains, New York, June 20-22, 1990.

Unfortunately, until recently the automatically generated attributes evaluators, as these programs are called, were too inefficient in time and/or space to honorably compete with their hand-written equivalents, unless the class of accepted AGs, and hence the expressive power, are severely restricted (see [7] for a good list of existing AG-processing systems).

Our goal in designing the FNC-2 AG-processing system [25] was hence to bring up-to-date technology and recent research results together into a production-quality system. Our motivation was definitely to bring a practical and usable solution to a problem which had mostly attracted theoretical interest. We wanted FNC-2 to be:

efficient: the generated evaluators should be as efficient in time and space as hand-written programs using the same basic data structures, in particular an explicitly-built tree to represent the input text (we ruled out from the start tree-less methods such as attributes evaluation during parsing for their very poor expressive power, see next item).

powerful: this efficiency should not be achieved at the expense of expressive power, i.e., FNC-2 should be able to process AGs in a very broad class, so that an AG-developer is not (too much) constrained in his design by the system.

easy to use: the input language of FNC-2 should enforce a high degree of programming safety, reliability and productivity.

versatile: the generated evaluators should allow to be interfaced with many other tools.

There exists (yet) no system which actually achieves these goals, and only a few which reach close to them. Among them we can quote GAG [32], developed by U. Kastens at Universität Karlsruhe, Linguist [13,16], developed by R. Farrow first at Intel, then at Columbia University, and now at Declarative Systems, and, although it is based on a somewhat different framework, the Synthesizer Generator [42,43,44], developed

by T. Reps and T. Teitelbaum at Cornell University. Comparisons between FNC-2 and these systems will appear in the relevant sections.

This paper presents the design decisions embodied in FNC-2 (section 2), various aspects of its implementation (section 3) and an account of preliminary experience we gained in using it (section 4). Future work is described in the final section.

2 Design Choices

2.1 Evaluation Methods

The first choice one must make when undertaking the design of an AG-processing system is that of the evaluation method(s) the generated evaluators will use. Indeed, this determines both the expected efficiency of the evaluators and the expressive power (accepted AG class) of the generator. A tremendous amount of research work has been devoted to the devising of evaluation methods that reach a good tradeoff between efficiency and expressive power (see [1,7,10] for good surveys).

2.1.1 Exhaustive evaluation

The requirement for FNC-2 to generate efficient evaluators ruled out methods based on dynamic scheduling: in addition to the execution of semantic rules, which is its basic job, an evaluator should do as little work as possible, which means that as much information as possible about the evaluation order should be embodied in the code of the evaluator itself and not computed at run-time. So we decided that FNC-2 would produce evaluators based on the visit-sequence paradigm [14,29]. Such an evaluator is a visit-sequence interpreter. There exist one visit-sequence per production, which is a sequence of instructions drawn from the following set:

`BEGIN` *i*: begin the *i*-th visit to the current node.

`EVAL` *s*: evaluate all the attributes in set *s*; the attributes on which elements of *s* depend are guaranteed to be available.

`VISIT` *i*, *j*: perform a recursive visit (the *i*-th one) to the *j*-th son of the current node; on that son, fetch the applied production and search the `BEGIN` *i* instruction in the corresponding visit-sequence.

`LEAVE` *i*: terminate the current (*i*-th) visit of the current node and return to its father; continue on the father with the instruction following the `VISIT` *i*, *j* which caused the current visit to begin.

It is clear that such evaluators can be implemented very efficiently.

It is possible to build visit-sequences for a given AG only if a total evaluation order can statically be determined, which is not the case for all AGs, but then this has the additional beneficial side effect that this order can also be used to improve the space consumption of the evaluators (see below).

The largest class of AGs for which such a total order exists is the *l*-ordered class [11], also called uniform in [14]. Unfortunately, on one hand this class is not as broad as could be dreamed of, and above all its characterization is an NP-complete problem [11]. There exist two possible approaches for solving this problem:

- restrict the accepted AG class to one more easily characterizable, such as the OAG class [29] for which there exists a polynomial test; however this decreases the expressive power;
- use a variant of the non-circular to *l*-ordered transformation [11] that applies to the strongly (i.e., absolutely) non-circular AG class [6] and is compatible with the visit-sequence paradigm [45].

We used the latter approach for FNC-2. This rather well-known transformation is described in [11,18,45]; it relies on the computation of *several* totally-ordered partitions of the attributes of each non-terminal, each order being a possible evaluation order for these attributes. The transformation proceeds as follows:

1. Compute the argument selectors (IO-graphs).
2. In top-down passes over the AG, compute for each non-terminal a set of totally-ordered partitions of its attributes. For each production:
   (a) pick a totally-ordered partition for the LHS non-terminal;
   (b) paste it together with the production dependency graph and the argument selectors of the RHS non-terminals;
   (c) topologically order the resulting graph; this determines on each RHS non-terminal a totally-ordered partition; memorize the mapping from the production number and partition for the LHS non-terminal to the partitions for the RHS non-terminals.

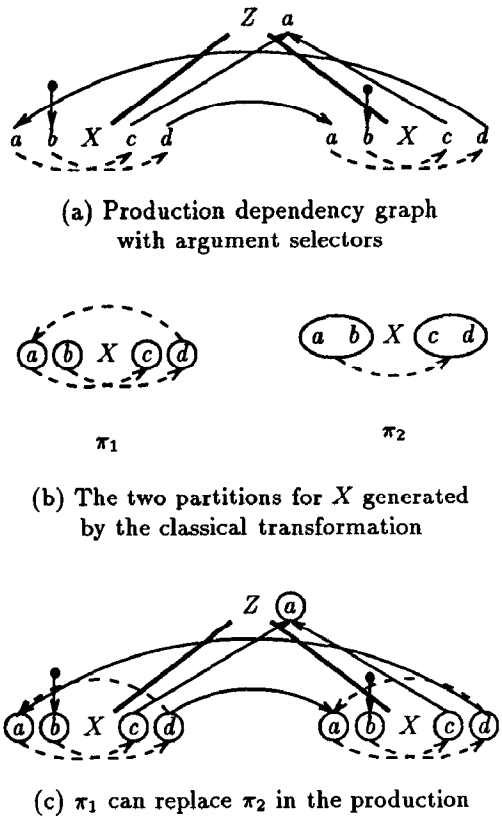
   Repeat until fixed point is reached.
3. The resulting AG has as non-terminals the pairs (old non-terminal, one of its partitions) and productions as determined by the above construction (memorization step). Note however that this AG needs not be explicitly built; rather, during the construction, a visit-sequence-based evaluator can be built, in which recursive `VISIT` instructions carry

an additional parameter that identifies the partition to use on the visited node (there exists one visit-sequence per pair (production, partition of its LHS non-terminal)).

This construction reaches its goal, in that the generated evaluators are completely deterministic, thus fast, while keeping the great expressive power of the SNC class. Unfortunately, it has exponential complexity, and the generated evaluators can be exponentially big (more precisely there can be an exponential number of totally-ordered partitions per non-terminal).

One of our works was to reduce this factor, by defining a coarser but still correct notion of equivalence of partitions which allows to strongly limit their proliferation [40]. The idea is similar to that of *covering* [37], a notion originally devised to reduce the complexity of non-circularity tests. In step (2c) of the above transformation, once you have determined a totally-ordered partition on the RHS non-terminals, and for each of them, check whether the newly-found partition can be replaced by an already existing one; here, "replace" means determining whether you can change the topological order so that it fits both that existing partition and the local dependencies (this is a necessary correctness condition), and also the partitions for the other RHS non-terminals (this condition is not strictly necessary but keeps the whole process polynomial)—see Figure 1. If no such "including" existing partition can be found, check whether the newly-found partition can replace one or more existing partitions in the productions in which they have been generated.

There exist several variations of this notion of equivalence of totally-ordered partitions (see [40] for more details and the formal proofs). With our "best" variation, dubbed *strong inclusion*, practical tests show that, on most practical AGs, we end up with exactly *one* partition per non-terminal,[1] when the classical transformation, with inclusion being the same as equality, ends up with sometimes more than five partitions per non-terminal on the average. This is clearly a good saving on the evaluators size. An additional benefit is that the running time of the transformation seems to be directly related to the total number of partitions computed for the whole AG, so that our optimized transformation runs much faster that the classical one and in almost-linear time. The only drawback of our scheme is that it tends to increase the number of visits to each node,[2] since a "replacing" partition must have at least as many distinct attributes sets as the replaced one (possibly more), and the number of visits is half that number of sets. However, on all the practical AGs we have used, this increase is less than 2% in average, and since pure tree-walking accounts only for a very small fraction of the evaluator running time (the rest is for the computation of the semantic rules), the dynamic effect is unnoticeable.



(a) Production dependency graph with argument selectors

(b) The two partitions for X generated by the classical transformation

(c) π_1 can replace π_2 in the production

Figure 1: Replacing a totally-ordered partition by another one

The evaluators generated by FNC-2 are hence as fast as those produced by GAG and Linguist, since they basically use the same paradigm. However FNC-2 is more powerful since it accepts a larger AG class.

2.1.2 Incremental evaluation

In addition, we provided for FNC-2 to be able to generate incremental attributes evaluators [42]. We devised a new incremental evaluation method [40] whose outline is as follows:

1. Generate an exhaustive evaluator which is able to start at any node in the tree. This is achieved by computing argument selectors as for the SNC class, but these selectors must be closed both "from below" and "from above" [18]. This construction hence applies only to a subclass of SNC AGs called doubly non-circular (DNC). This class is however

[1] This does not mean that we have found a polynomial test for characterizing the l-ordered class: there exist l-ordered AGs for which our transformation will end up with more than one partition per non-terminal.

[2] Note that finding the set of partitions achieving the minimum number of visits is yet another NP-complete problem.

larger than the l-ordered class, and our SNC to l-ordered transformation allows to actually use this method for any SNC AG. Also, the DNC class is characterizable in polynomial time. This evaluator is efficient since it is completely deterministic.

2. Add to this exhaustive evaluator a set of "semantic control" functions allowing to limit the reevaluation process to affected instances. This control is based on the determination of the status of each attribute instance (changed, unchanged or unknown) and the comparison of its old and new values [42]. Note that the notion of equality used in this comparison can be adapted to the problem at hand, which enhances versatility.

Parigot [40] also describes how this method can accommodate multiple subtree replacements.

Reps [42] proposes two incremental evaluation methods, one for (plainly) non-circular AGs and one for ordered AGs (see also [49,50]). Our method should prove more efficient than the former and as efficient and more powerful than the latter. We believe that it is quite attractive, in particular because of its versatility, but this must be confirmed by practical experiments that we have not yet conducted.

2.2 Space Management

As demonstrated by the works of Kastens [30,31,32] and others [12,17], the necessity to have a statically-determinable total evaluation order to produce visit-sequence-based evaluators has the beneficial side-effect to allow to conduct very fine static analysis of the lifetime of each attribute instance, which in turn allows to determine the most efficient way to store it: either in a global variable, or on a global stack, or, as a last resort, at tree nodes. Of course we have integrated these works in FNC-2 (for exhaustive evaluation).

Furthermore we have improved these works in several respects [27,28]:

- We have relaxed the unnecessarily strict conditions that Kastens [30,32] imposed on stack management, namely that only the top element can be accessed. We allow accesses below the top of stack, provided that the depth can be computed statically each time it is needed to do so. This, combined with artificial delaying of `POP` instructions, allows to store in stacks *every* temporary attribute (which typically account for more than 80% of all attributes).

- With a formalism that is both finer than that of [30,32] and simpler than that of [31], we are able to store more temporary attributes in global variables. Technically, this is because, when statically simulating the execution of the evaluator by means of the *grammar of visits*, we take into account the visit numbers, which Kastens did not do.

- Along the lines of [12], but with a much simpler formalism based on the *grammar of visits and contexts*, we are able to determine when a non-temporary attribute can be stored in a global variable.

- We have improved Kastens' technique for packing together several global variables or several stacks. He used, to decide each such grouping, a mere feasibility criterion, which on one hand was far from optimal and, on the other hand, was sensitive to the textual order of the productions and semantic rules in the AG. Our criterion is based on the number of copy rules a potential grouping would eliminate. Not only does it reduce memory consumption in a purely declarative manner but it also reduces the running time, especially when it suppresses stack management operations. It is well-known that solving this problem in an optimal way is NP-complete, so we use heuristics, based on a static traversal of the visit-sequences, to keep an acceptable complexity.

Note that when we say that we have improved over Kastens' work, we must recognize that we do not refer to his latest work [31], in which he reaches nearly the same results as we (except for the packing algorithm). However we did most of our work independently from him; in addition, we believe our formalism is simpler than his.

Presently we work on the last category of attributes, namely the non-temporary ones which cannot be stored in global variables. First, it seems possible to use the grammar of visits and contexts, or a variant thereof, to determine whether a non-temporary attribute can be stored in a strict stack, i.e., with accesses only to the top element and without trying to extend the lifetimes. Second, we are working on a special data structure, similar to so-called "cactus stacks," to hold attributes failing even this latter test, and on the corresponding lifetime extensions. We are quite confident that, in the end, we'll be able to store *all* the attributes out of the tree. This will of course improve storage consumption, but the most important consequence is a yet to be estimated, but surely great, increase in the expressive power of AGs. Indeed, since with that scheme the only purpose of the tree is to conduct the evaluator, it needs not be a physical object any more; it only has to *mimic* a physical tree. For instance, attributes evaluation on DAGs (i.e., trees with shared subtrees) comes for free. Other applications need yet to be explored.

Practical results are given in section 4.

2.3 Modularity

AGs have many advantages as a specification method but their most stringent drawback in this respect is their absence of modularity, i.e., the necessity to program a complete application as a single, monolithic AG. As applications get larger and more complicated, this scheme becomes more and more unworkable; a striking example is the ADA front-end developed using the GAG system [48]: it is a single AG of more than 500 *pages*! We clearly had to address this issue if we wanted FNC-2 to have a chance to become a production-quality system.

Another problem with this approach is that it forces to use a single formalism to tackle a whole application, whereas it can be preferable to use different specialized frameworks for solving different subproblems. There are several instances of this situation in compiler construction. First, it is *a priori* impossible to use AGs for fixed-point-like computations, such as those involved in data flow analysis (although [46] shows they are not always necessary, and [15] shows how to implement them through an extension of the AG framework). Secondly, the AG formalism imposes that the representation of the program, i.e., the input tree, does not change at all during the whole computation; this is clearly not well suited for e.g. optimization phases, which involve transformations of the program representation.

So we had to devise a scheme allowing to build an application as a set of modules, some of them being specified by AGs and the others by other techniques. This is achieved by considering that an AG specifies, and an evaluator implements, an (attributed) tree to attributed tree mapping. More precisely, each evaluator takes as input a tree, possibly decorated with a collection of attributes, and produces as output one or more decorated trees. This allows to:

1. decompose what used to be specified by a single AG into smaller modules, each of them specified by a smaller and hence more easily manageable AG;
2. interface the corresponding evaluators with other modules, providing that the latter be also based on the tree-to-tree mapping paradigm (this is for instance the case for tree transformation systems such as those generated by OPTRAN [36]).

Our scheme is actually a variation of attribute coupled grammars [20]. This formalism is interesting since descriptional composition [20,23] allows to combine several "modular" AGs into a single efficient evaluator which does not build the intermediate trees. We already know that this process is feasible with FNC-2, in particular because the SNC class is closed under descriptional composition [23], but we have not yet implemented it.

None of the systems used as reference points offer true modularity. They all provide for externally-defined semantic functions, which is a form of modularity, but it lies outside the AG formalism.

2.4 Input Language

Many works have been concerned with the construction of systems producing efficient evaluators from an AG [7], but surprisingly few of them have tried to design specialized AG-description languages. Actually, there are two classes of systems:

1. those offering such a specialized language;
2. those using as input language their implementation language merely augmented with constructions to define and access attribute occurrences.

Of course, some systems are a mixture of both classes...

We have shown [24] that the second approach has many drawbacks that would hinder nearly all of the FNC-2 design requirements. Hence we have decided to design a new language specifically devoted to the description of AGs. The requirements for this new language were: easiness of use; great expressive power; enforcement of programming safety and reliability; versatility; readability and modularity; closeness to Knuth's theory of AGs; independence from the evaluation method and the implementation languages, but easy translation to those languages; opportunities for optimizations; simplicity. Our language, called OLGA [25], fulfills these goals by providing the following features:

- OLGA is purely applicative, but not functional.
- It is strongly typed, with polymorphism, overloading and type inference.
- It is block-structured and modular. Function definitions are blocks and can be nested. Compilation units are declaration and definition modules, possibly parametrized, and AGs; as said above, an AG defines a tree-to-tree mapping (there are tree types and tree construction functions). An AG can be structured into phases, each of them containing productions together with a number of semantic rules; a given production may appear in several phases or not at all. Phases and productions are blocks; a value local to a production and depending on some attributes is hence a local attribute. Declaration modules export the entities they contain, which can be opaque. Each block can then import some or all of those entities, and can contain local declarations.
- OLGA provides modern constructs such as pattern matching, exception handling, display of error messages, construction of circular structures, ...

- The syntactic base of an AG written in OLGA is an abstract syntax rather than a concrete one. The formalism is quite close to Metal [4] and provides for list nodes. There exist special constructs for defining and accessing attributes on list productions.
- Most copy rules can be automatically generated and need not be specified explicitly. In addition, you can specify a set of attribute classes and semantic rules models defining them; the system will automatically instantiate these models into actual semantic rules whenever necessary and applicable [35] (this is analogous to what is described in [9, 47]).

The three reference systems also offer each a specialized input language (but they are not representative in this respect!). Although it is hard to compare the "efficiency" and "power" of programming languages, it seems that OLGA is at least as good as the others (see also section 4); [24] gives a rather detailed presentation of OLGA and compares it with Aladin, SSL and the Linguist language.

3 Implementation

The FNC-2 system is composed of three parts, linked through well-defined interfaces: the OLGA front-end, the evaluator generator and the translators (see Figure 2). This structure allows for instance to use the evaluator generator as a stand-alone processor, through a crude interface. In addition, FNC-2 comes with a number of companion processors.

3.1 The Evaluator Generator

The evaluator generator (see Figure 3) is the "engine" of the system, in which all the fundamental knowledge about attributes evaluation is concentrated. It receives as input an abstract AG (syntax and local dependencies) and produces an abstract evaluator (visit sequences with calls to evaluate the semantic rules, memory map and storage manipulation operations).

The first step in the construction is the SNC test. The whole process aborts if the AG fails this test, but then an interactive circularity trace system [39] allows to easily discover the origin of the failure; this allows to take full advantage of the power of the SNC class. Then the DNC test is performed; if it succeeds, the OAG test is performed. Actually there exists an infinity of incomparable OAG(k) classes [3]; by default FNC-2 performs the OAG(0) test but can be directed to test for the OAG(k) class for any given k.

If either the DNC or OAG test fails, the SNC to l-ordered transformation is invoked. Note that even if the OAG test fails, the information computed by the DNC test is used by the transformation step to run faster and produce better results. Also note that cascading these phases costs the same as performing the OAG test from scratch, since the first phase of the OAG test is the DNC test, and the first phase of the latter is the SNC test.

The next step is the production of the visit-sequences from the total orders computed either in the OAG test or in the transformation. The final step is the space optimization, which produces the memory map and enriches the visit-sequences with storage manipulation operations.

During the development of an AG, it is also possible to save some time by producing non-deterministic visit-sequences directly after the SNC test. In this case, no space optimization is attempted since no total evaluation order is available.

All the algorithms cited above belong to the Grammar Flow Analysis framework [38] and make heavy use of our improvements to this technique [26]. Thus the evaluator generator is quite fast: an evaluator for a complete Pascal to P-code compiler is produced in less than two minutes on a Sun-3/60 (see also the figures in Table 1).

3.2 The OLGA Front-end and Translators

These components are of course bootstraped, i.e., they are specified by AGs written in OLGA and automatically generated by FNC-2 itself. We make full use of the provision for modularity (see the figures in next section).

OLGA is a big language, and its analysis and implementation are hard tasks, even with AGs. Presently not all of the language is implemented; either the front-end or the translators reject some valid programs. The most notable omissions are full polymorphism, parametrized modules and exceptions.

Presently there are two translators, one to C and one to Lisp. Both are rather naïve; this is especially noticeable for the C implementation, which lacks a garbage collector. However both are usable, the best evidence being the bootstrap of the system. These translators are preceded by a common optimizer, which in particular performs tail recursion elimination and builds deterministic decision trees for the OLGA pattern-matching construct.

3.3 The Companion Processors

FNC-2 comes with several companion processors. *Asx* analyses attributed abstract syntaxes descriptions, which play a great role in our formalism since they describe the input and output data of the evaluators (attributed trees). *Atc* generates abstract tree constructors which run in parallel with, and are driven by, parsers

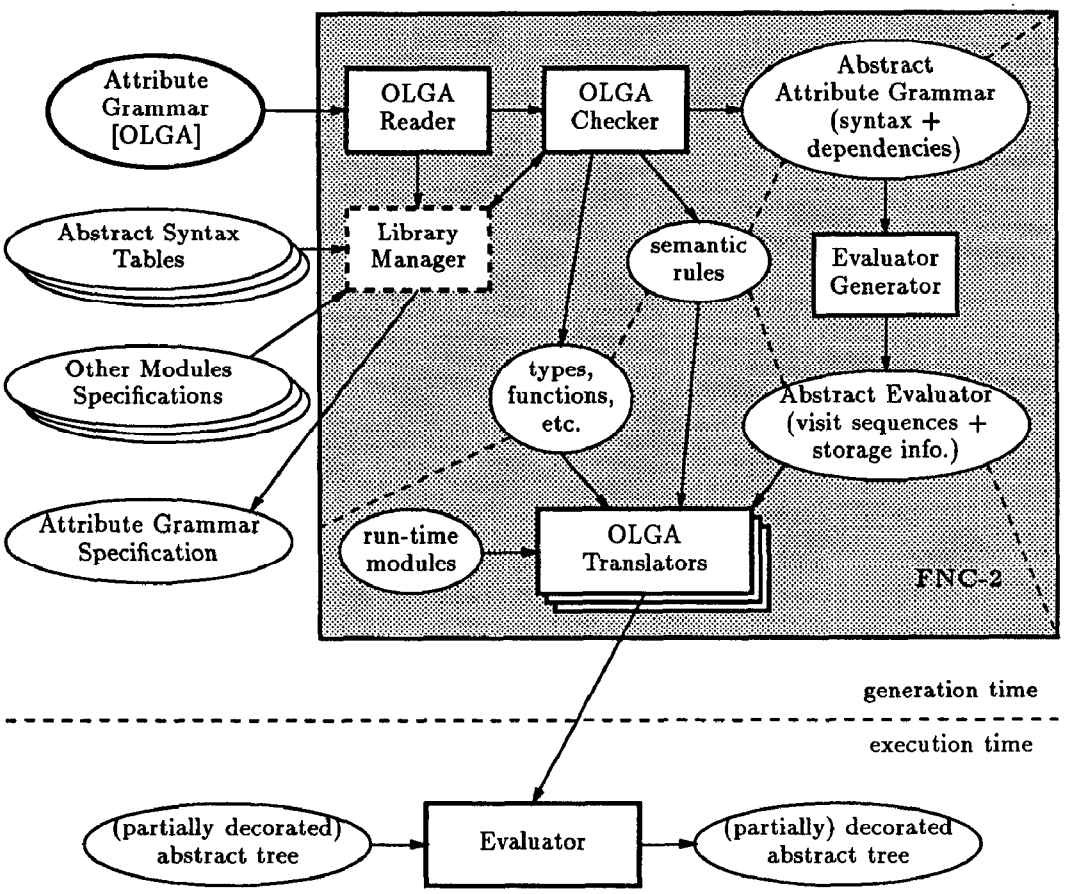


Figure 2: The FNC-2 system

constructed by the SYNTAX system [5]; another version of *atc* built on top of Yacc is available. *Ppat* generates unparsers for attributed abstract trees. *Mkfnc2* automates the construction of complete applications using FNC-2 and the other processors; it is a pale start at the library manager depicted in Figure 2.

All those subsystems use as input language an "extended subset" of OLGA. Of course, they are bootstraped.

4 Evaluation

The first hand-written version of FNC-2 was completed in early 1989 and immediately bootstraped. Since then development continues with OLGA and its companion languages, however their main applications are still the system components themselves. Two realistic "external" applications are under development:

- a compiler from the PARLOG parallel logic programming language to code for the SPM abstract machine [22];
- a compiler from full ISO Pascal to P-code.

In addition, FNC-2 is used in the PAGODE code generator generator [8], both to implement the system itself and as the back-end for one of its phases.

In spite of this rather long list, the experience reported below is preliminary and rather incomplete.

Since the above-described applications are still under development, the statistics below will use as basic data the AGs and modules specifying parts of the system itself.

4.1 The Evaluator Generator

Table 1 presents some figures gathered from the execution of the evaluator generator (as part of FNC-2) on various AGs. These AGs specify the following parts of the system:

1. The construction of the module dependency graph in *mkfnc2*.
2. The test for well-definedness of an AAS specification in *asx*.
3. The translation to C of the tree construction part of an *atc* specification.

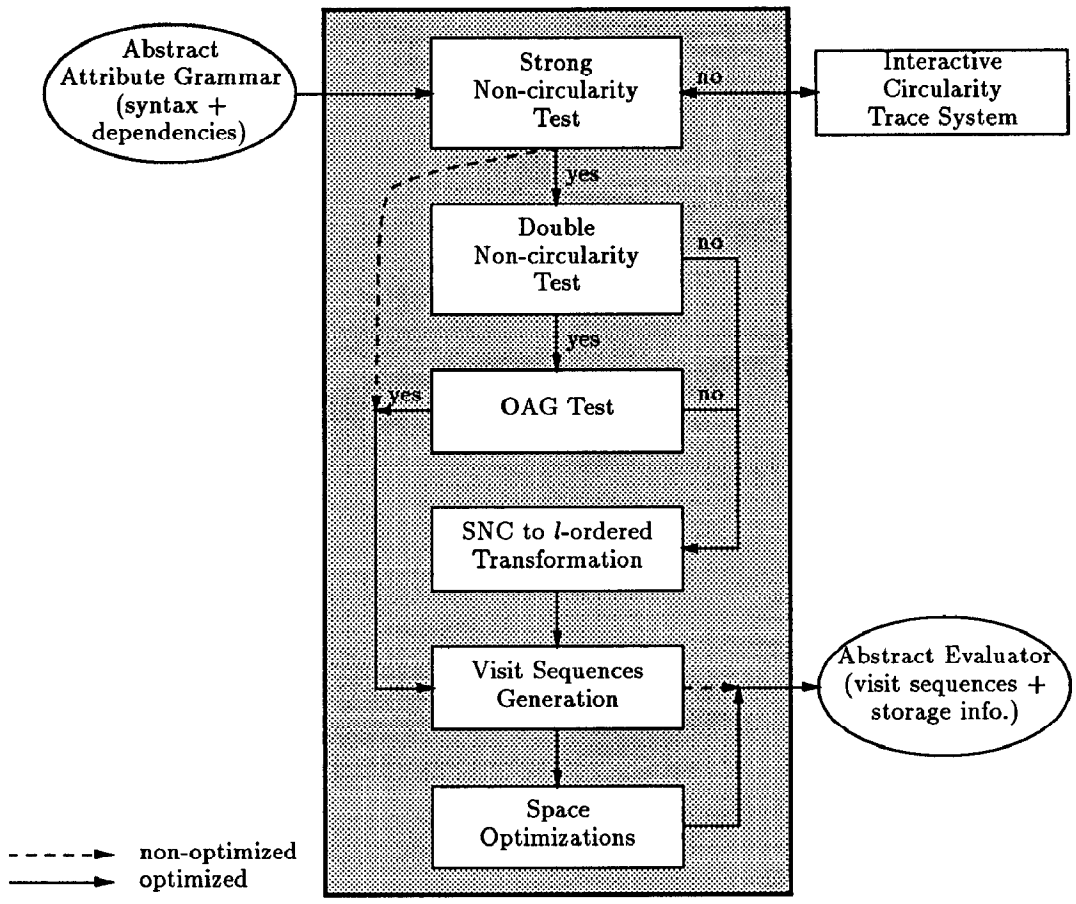


Figure 3: The evaluator generator

AG	1	2	3	4	5	6	7	ave.
# phyla	6	12	35	35	74	74	74	
# operators	27	47	131	131	319	319	319	
# occ. attr.	20	98	94	320	992	72	249	
# sem. rules	99	417	444	1420	5318	451	1497	
class	OAG(0)	OAG(0)	OAG(0)	OAG(0)	DNC	OAG(0)	OAG(1)	
% variables	80	86	33	61	60	48	35	56.5
% stacks	20	7	62	35	28	52	64	36
% non-temp.	0	7	5	4	12	0	1	7.5
# variables	4	25	12	44	106	7	26	
# stacks	3	5	23	30	49	11	47	
% elim./copy	65	64	20	20	30	23	16	27
% elim./poss.	74	88	89	84	94	97	91	91
time	1.78	9.03	10.00	45.31	489.41	10.37	48.23	

Table 1: Statistics gathered for the evaluator generator

4. The type-checking of the tree construction part of an *atc* specification.

5. The type-checking of an OLGA specification, be it an AG or a module, and test for well-definedness of an AG. This is the largest and most significant example in the set.

6. The test for tail-recursive functions in an OLGA specification (the construction of deterministic decision trees for pattern-matching is not yet part of the production version of the system).

7. The translation to C of the non-AG parts (types, functions, semantic rules, etc.) of an OLGA specification.

Note that these AGs are not self-contained: they import many types, functions and other entities from separate modules.

The top part of Table 1 gives an idea of the size of each AG. Phyla and operators correspond roughly to non-terminals and productions in a concrete grammar. The number of attribute occurrences is the sum on all phylas of the number of attributes attached to each.

The so-called class of an AG is the smallest class to which it belongs, as determined by the evaluator generator. AG 5 is not OAG(k) for any k, which shows that it is advantageous to have a system accepting a class larger than OAG. We don't know whether it is truly l-ordered: our SNC to l-ordered transformation ends up with 1.03 totally-ordered partitions per non-terminal in average (max. 2). On the same AG, the classical transformation using equality ends up with 4.15 partitions per non-terminal in average (max. 29)! AG 7 is not OAG(0) but is OAG(1). This was discovered by trial and error and leads to better results for space optimization than our transformation (this is not always the case); however this trial and error process should be performed only in the very final phase of the development of an AG.

The next part of Table 1 deals with space optimization. The first three figures give the proportion of attribute occurrences which are stored in global variables, global stacks or at tree nodes; the latter class corresponds to non-temporary attributes because we have not yet implemented the test for storing non-temporary attributes in global variables. The average figures in the last column are weighted by the number of attribute occurrences in each AG, which is a meaningful indication of its size. These figures are static, i.e., they refer to the AG itself rather than to its dynamic execution. Dynamic measures [28] show a decrease of the number of attribute storage cells by a factor of 4 to 8 in the execution of AG 5 on various source texts. These figures are already quite good and will be even better when our space optimization scheme is fully implemented. The next four figures report the effect of our grouping algorithm. On AG 5 for instance, it cuts the number of global variables from 595 down to 106 and the number of global stacks from 278 to 49; however the dynamic effect on storage consumption is much less noticeable. More important is the next figure, which describes the proportion of copy rules which have been eliminated w.r.t. the total number of copy rules; again, this is a static figure. It may seem low, but it must be noticed that not all copy rules can be eliminated: for instance, storing two different attribute occurrences in the same variable makes the evaluator incorrect when both are live at the same time. The next figure—the proportion of rules that have *actually* been eliminated w.r.t. those which could *theoretically* be eliminated—show that we reach close to the optimum (recall that achieving the optimum is NP-complete).

The last figure is the CPU time on a Sun-3/60 machine. The whole process is clearly non-linear but also non-exponential. We believe that the efficiency of the evaluator generator is reasonable given the extensive optimizations it performs.

4.2 The Generated Evaluators

As examples of evaluators generated by FNC-2 we have chosen some parts of FNC-2 itself, namely the type-checking of the OLGA source text and the translation to C of its non-AG parts. Table 2 summarizes the execution of FNC-2 on the above-described AGs while Table 3 is for some modules, i.e., OLGA texts not specifying an AG (Cn are declaration modules and Fn are the corresponding definition modules).

In both tables, the first row gives the length of the input text. The second to fourth give the CPU time, in seconds, for some phases of its processing (measured on a Sun-3/60 machine). "Input" includes scanning, parsing and initial tree construction; this phase is generated by *atc* and SYNTAX. "Typing" is the type- and well-definedness checking, as specified by AG 5 above, and "translator" is the translation to C of the non-AG parts (AG 7 above)—translation to C of the visit-sequences is presently hand-written. There are other phases involved in the processing, e.g. evaluator generation, but these last two phases correspond exactly to evaluators generated by FNC-2, which is what we are interested in here. "Memory" is the size, in kbytes, of the memory used by the whole process, as reported by the *time* command built in *csh*. "Time" is the total CPU time including all the phases (in particular the evaluator generation for Table 2).

The last column reports the average speed, in lines per minute, of the various phases and the whole process. The average global speed in Table 2 is not meaningful since it includes the evaluator generation, a non-linear

AG	1	2	3	4	5	6	7	speed
# lines	354	674	1098	1465	3212	1232	2083	
input	1.22	2.05	3.58	4.98	11.83	2.63	7.48	17940
typing	2.17	5.98	15.88	23.42	86.40	31.72	46.00	2820
translator	1.82	3.00	5.20	7.42	18.03	4.13	11.22	11940
memory	680	1048	1328	2104	4584	1528	2752	
time	10.3	37.8	68.1	141.8	914.5	101.6	225.3	402

Table 2: Statistics gathered for the FNC-2 system (on AGs)

module	C1	F1	C2	F2	C3	F3	C4	F4	C5	F5	C6	F6	speed
# lines	189	372	320	3188	268	1083	390	1186	391	905	86	268	
input	0.70	1.53	1.07	13.47	1.07	4.62	1.50	5.02	1.70	4.53	0.17	1.15	14300
typing	1.58	3.32	4.77	51.73	4.08	13.80	6.22	14.80	5.00	14.72	0.40	1.72	4315
translator	0.85	1.98	0.28	16.62	0.85	5.93	2.23	5.90	2.43	6.20	0.33	1.38	11490
memory	480	632	848	2912	568	1032	656	1072	744	1080	384	536	
time	6.28	12.30	38.03	191.51	12.32	36.30	16.58	41.43	22.15	41.13	1.55	5.82	1235

Table 3: Statistics gathered for the FNC-2 system (on modules)

phase; on the other hand, the results in Table 3 are typical of a compiler-like application. The difference in the speed of the typing phases in Tables 2 and 3 stem from the fact that, in addition to mere type-checking which is the same as for a module, an AG needs to be checked for well-definedness; furthermore, this phase is responsible for constructing the "abstract AG" to be input to the evaluator generator. The difference in the speed of the input phases stems from the fact that modules are, in average, substantially smaller than AGs, so that "constant overhead" (initialization, etc.) is more noticeable in the former case. Memory consumption of the whole process averages to between 1.3 and 1.4 kbytes per line of input and does not vary much whether the evaluator generator is invoked or not; this means that most of the space is used for data created for or by the evaluators. As a comparison point, we have—not very precisely—measured the behavior of Sun's C compiler (no optimization nor assembly): average speed amounts to 5800 l/mn and memory consumption to .09 kb/l. However it must be kept in mind that it is a one-pass compiler which does not build a complete representation of the input program.[3]

We believe that the efficiency of the generated evaluators is already rather satisfactory: comparison between the hand-written version of the system and the bootstraped version shows that the latter is only between two and four times slower on average; comparison with the C compiler shows a speed ratio of five, but its structure makes it a somewhat unfair comparison point (see our design goals in section 1). Finer analysis shows that this slowdown must not be attributed to the evaluator as such but to the execution of the semantic rules themselves, because our OLGA to C translator is really naïve. Hence we are quite confident that we can improve this ratio to make it close to unity.

The same applies to space consumption: the attribute storage cells themselves account only for a small fraction of the data space, the rest being used by the (undecorated) tree and above all by the values of complex attributes (e.g. symbols tables). The latter will be strongly reduced when we implement a garbage collector and use more sophisticated analysis techniques on OLGA to improve sharing and replace some constructive modifications by destructive updates.

4.3 Use of the System

Our experience shows that the expressive power of FNC-2, i.e., the SNC class, is very useful: many AGs we have written were, at some time during their development, not ordered and not even *l*-ordered. Of course it would have been rather easy to make them actually ordered, but using FNC-2 gave us more freedom in our development. Also, a great expressive power is quite useful when AGs are automatically produced by other systems, such as the above-cited PAGODE: designers of these systems can ignore "mundane" issues such as evaluation order. We also would like to point out that AGs are, to the best of our knowledge, the only method which really supports incremental development: you may freely add new attributes and semantic rules, and then test your AG without having to completely specify it.

Our experience with OLGA is also quite satisfactory, and we find it a very good specification tool: although

[3] A tree is built for each expression, but it is immediately translated into target code and the space it occupies is reused for the next expression, so the compiler can roughly be considered as to run in constant space.

	# files	# lines			
		min	max	total	ave.
AGs	7	354	3212	10118	1445
AASes	8	8	381	779	97
decl. mod.	15	28	391	2891	193
def. mod.	15	55	3188	13404	894
atc	4	60	2089	2575	644
total	49	8	3212	29767	607

Table 4: Source files in the FNC-2 system

it is hard to measure the productivity of a programmer, knowing that the complete development of the system is the one-year work of only one full-time researcher plus one or two part-time students gives an idea of the easiness of use of OLGA. When we compare our experience with OLGA to that we gained with an earlier system using Lisp as base language, we are sure that we have made the right choice (see the discussion at the beginning of section 2.4), even though the present implementation of OLGA is rather inefficient. We prefer to spend our efforts in improving this efficiency rather than switching back to an existing implementation language. Furthermore, using a specialized AG-description language allows to translate it to several implementation languages, which we think is an important advantage for versatility. There remains to implement the missing features, which would improve again the easiness of use of OLGA, and above all improve its implementation. In addition, we are thinking of simplifying it by unifying the syntactic and semantic domains, that is, the trees and the attributes, as is done in SSL [43,44] or MARVIN [21]; after all, a grammar specifies the same structures as record, union and list types.

If we had to single out a specific advantage of OLGA and FNC-2 over their competitors, this would undoubtedly be their provisions for modularity. Table 4 shows the organization of the source code of the system (excluding *ppat*, still under development, and the modules written in C, namely the evaluator generator and the run-time system). If all this code was gathered in a single file, or even one file per subsystem, it would be impossible to manage.

Modularity also eases the reuse of modules in several applications. As an example, Figure 4 shows the organization of the *ppat* subsystem: ellipses represent modules written in OLGA, *asx* or the *ppat* language itself. Solid arrows represent flow of information: outside a shaded box this corresponds to an importation; inside a shaded box this represents an internal representation in the form of an attributed abstract tree (a dotted arrow connects the corresponding *asx* specification to the arrow representing the tree); from an ellipse to a white box this corresponds to some generation process. Files X`.asx` and X`.ppat` are written by the user; they respectively specify the input attributed abstract trees and their textual representation. File X`-ppat.olga` is generated from those by *ppat* and then combined with files `boxes*` to form the actual unparser. The figure shows that most of the unparser is independent from the input tree language and that the dependent part is hence easier to generate.

As said previously, none of the other AG systems allows to develop an application in a modular way.

5 Conclusion and Future Work

We have presented the FNC-2 attribute grammar system, which aims at production-quality by providing efficiency, expressive power, ease of use and versatility. We are still far from our final goal but preliminary experience shows we are on the right track.

Future work will concentrate in improving the implementation of OLGA and interfacing FNC-2 with many other systems. As for the former point, Appel's work on a portable implementation of a run-time system for ML, including a garbage collector [2], is a very good starting point; similar work addressing Lisp implementation is in progress at INRIA. As for the latter point, the OLGA to Lisp translator is the basis of the integration of FNC-2, CENTAUR [4] and GIGAS [19] into a powerful and attractive competitor to the Synthesizer Generator; in addition, this will serve as a testbed for our incremental evaluation algorithm and for application-specific propagation termination conditions.

Another aspect of our future work is the use of FNC-2 to develop highly specialized languages and systems to develop (parts of) compilers. Indeed, AGs are not specialized for compiler construction, and they do not embody any specific expertise for this task. It seems possible to embody this expertise in specialized systems, as demonstrated by the works of Reiss [41] for identification and symbol table management, and those of Kildall [33] for data flow analysis (see also [46] for how to use AGs for this task). Then our dream of a complete and usable compiler development workbench would become a reality.

References

1. Alblas, H. Attribute Evaluation Methods. Memorandum INF-89-20, Onderafdeling der Informatica, Tech. Hogeschool Twente, 1989.
2. Appel, A. W. A Runtime System. Draft, Princeton Univ., Feb. 1989.
3. Barbar, K. Classification des grammaires d'attributs ordonnées. Rapport 8412, Univ. de Bordeaux I, Apr. 1984.

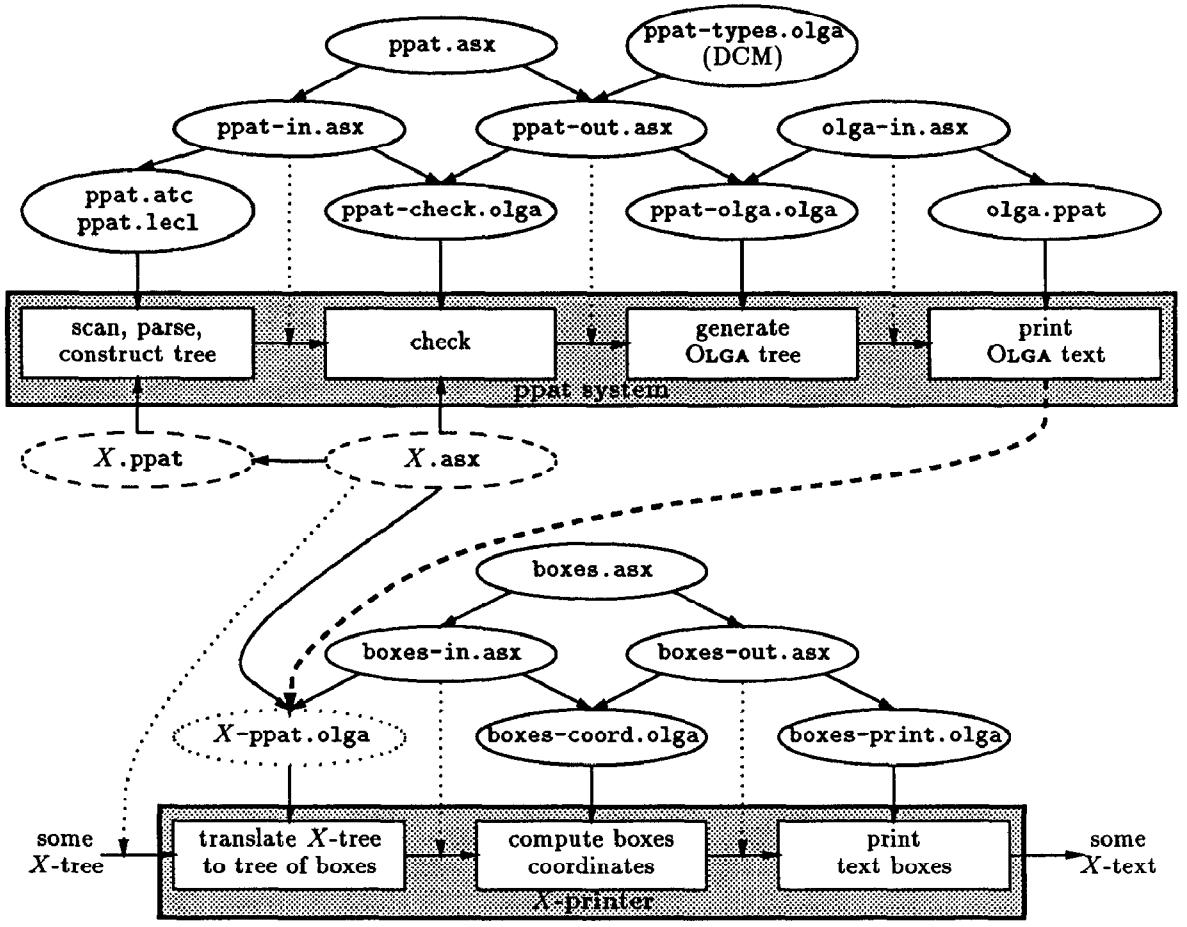


Figure 4: The *ppat* subsystem

4. Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B. and Pascual, V. CENTAUR: the System. In *ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments* (Boston, MA, Nov. 1988). *SIGSOFT Software Eng. Notes 13*, 5 (Nov. 1988), 14–24. Joint issue with *ACM SIGPLAN Notices 24*, 2 (Feb. 1989).
5. Boullier, P. and Deschamp, P. *Le système SYNTAX—Manuel d'utilisation et de mise en œuvre sous Unix*. INRIA, Rocquencourt, Sept. 1988.
6. Courcelle, B. and Franchi-Zannettacci, P. Attribute Grammars and Recursive Program Schemes. *Theoret. Comput. Sci. 17*, 2 and 3 (1982), 163–191 and 235–257.
7. Deransart, P., Jourdan, M. and Lorho, B. *Attribute Grammars: Definitions, Systems and Bibliography*. Lect. Notes in Comp. Sci., vol. 323, Springer-Verlag, New York–Heidelberg–Berlin, Aug. 1988.
8. Despland, A., Mazaud, M. and Rakotozafy, R. Using Rewriting Techniques to Produce Code Generators and Proving them Correct. Rapport RR-1046, INRIA, Rocquencourt, June 1989. To appear in *Sci. Comput. Programming*.
9. Dueck, G. D. P. and Cormack, G. V. Modular Attribute Grammars. Research report CS-88-19, Univ. of Waterloo, May 1988.
10. Engelfriet, J. Attribute Grammars: Attribute Evaluation Methods. In *Methods and Tools for Compiler Construction*, B. Lorho, Ed. Cambridge Univ. Press, Cambridge, 1984, pp. 103–138.
11. Engelfriet, J. and Filè, G. Simple Multi-Visit Attribute Grammars. *J. Comput. System Sci. 24*, 3 (June 1982), 283–314.
12. Engelfriet, J. and de Jong, W. Attribute Storage Optimization by Stacks. Rapport 88-30, Vakgroep Informatica, Rijksuniv. te Leiden, Dec. 1988. To be published.
13. Farrow, R. Generating a Production Compiler from an Attribute Grammar. *IEEE Software 1*, 4 (Oct. 1984), 77–93.
14. Farrow, R. Sub-Protocol-Evaluators for Attribute Grammars. In *ACM SIGPLAN'84 Symp. on Compiler Construction* (Montréal, June 1984). *ACM SIGPLAN Notices 19*, 6 (June 1984), 70–80.

15. Farrow, R. Automatic Generation of Fixed-point-finding Evaluators for Circular, but Well-defined, Attribute Grammars. In *ACM SIGPLAN'86 Symp. on Compiler Construction* (Palo Alto, CA, June 1986). *ACM SIGPLAN Notices 21*, 7 (June 1986), 85–98.

16. Farrow, R. *The Linguist Translator-writing System—User's Manual* version 6.25. Declarative Systems Inc., Palo Alto, CA, June 1989.

17. Farrow, R. and Yellin, D. M. A Comparison of Storage Optimizations in Automatically-Generated Attribute Evaluators. *Acta Inform. 23*, 4 (1986), 393–427.

18. Filè, G. Classical and Incremental Attribute Evaluation by Means of Recursive Procedures. *Theoret. Comput. Sci. 53*, 1 (Jan. 1987), 25–65.

19. Franchi-Zannettacci, P. Attribute Specifications for Graphical Interface Generation. In *Information Processing '89* (San Francisco, CA, Aug. 1989), G. X. Ritter, Ed. North-Holland, Amsterdam, pp. 149–155.

20. Ganzinger, H. and Giegerich, R. Attribute Coupled Grammars. In *ACM SIGPLAN'84 Symp. on Compiler Construction* (Montréal, June 1984). *ACM SIGPLAN Notices 19*, 6 (June 1984), 157–170.

21. Ganzinger, H., Giegerich, R. and Vach, M. MARVIN: a Tool for Applicative and Modular Compiler Specifications. Forschungsbericht 220, Fachbereich Informatik, Univ. Dortmund, July 1986.

22. Garcia, J., Jourdan, M. and Rizk, A. An Implementation of PARLOG Using High-Level Tools. In *ESPRIT '87: Achievements and Impact* (Brussels, Sept. 1987), Commission of the European Communities—DG XIII, Ed. North-Holland, Amsterdam, pp. 1265–1275.

23. Giegerich, R. On the Relation between Descriptional Composition and Evaluation of Attribute Coupled Grammars. Forschungsbericht 221, Fachbereich Informatik, Univ. Dortmund, July 1986.

24. Jourdan, M., Le Bellec, C. and Parigot, D. The Olga Attribute Grammar Description Language: Design, Implementation and Evaluation. In *Workshop on Attribute Grammars and their Applications (WAGA)* (Paris, Sept. 1990). Lect. Notes in Comp. Sci., Springer-Verlag, New York–Heidelberg–Berlin.

25. Jourdan, M. and Parigot, D. *The FNC-2 System User's Guide and Reference Manual.* INRIA, Rocquencourt, Feb. 1989. This manual is periodically updated.

26. Jourdan, M. and Parigot, D. Techniques for Improving Grammar Flow Analysis. In *ESOP '90* (Copenhagen, May 1990).

27. Julié, C. Optimisation de l'espace mémoire pour l'évaluation des grammaires attribuées. Thèse, Dépt. d'Informatique, Univ. d'Orléans, Sept. 1989.

28. Julié, C. and Parigot, D. Space Optimization in the FNC-2 Attribute Grammar System. In *Workshop on Attribute Grammars and their Applications (WAGA)* (Paris, Sept. 1990). Lect. Notes in Comp. Sci., Springer-Verlag, New York–Heidelberg–Berlin.

29. Kastens, U. Ordered Attribute Grammars. *Acta Inform. 13*, 3 (1980), 229–256.

30. Kastens, U. The GAG-System—A Tool for Compiler Construction. In *Methods and Tools for Compiler Construction*, B. Lorho, Ed. Cambridge Univ. Press, Cambridge, 1984, pp. 165–182.

31. Kastens, U. Lifetime Analysis for Attributes. *Acta Inform. 24*, 6 (Nov. 1987), 633–652.

32. Kastens, U., Hutt, B. and Zimmermann, E. *GAG: A Practical Compiler Generator.* Lect. Notes in Comp. Sci., vol. 141, Springer-Verlag, New York–Heidelberg–Berlin, 1982.

33. Kildall, G. A unified approach to global program optimization. In *1st ACM Symp. on Principles of Progr. Languages.* ACM Press, New York, NY, Jan. 1973, pp. 194–206.

34. Knuth, D. E. Semantics of Context-free Languages. *Math. Systems Theory 2*, 2 (June 1968), 127–145.

35. Le Bellec, C. Spécification de règles sémantiques manquantes. Rapport de DEA, Dépt. d'Informatique, Univ. d'Orléans, Sept. 1989.

36. Lipps, P., Möncke, U. and Wilhelm, R. OPTRAN – A Language/System for the Specification of Program Transformations: System Overview and Experiences. In *Compiler Compilers and High Speed Compilation* (Berlin, Oct. 1988), D. Hammer, Ed. Lect. Notes in Comp. Sci., vol. 371, Springer-Verlag, New York–Heidelberg–Berlin, pp. 52–65.

37. Lorho, B. and Pair, C. Algorithms for Checking Consistency of Attribute Grammars. In *Proving and Improving Programs* (Arc et Senans, July 1975), G. Huet and G. Kahn, Eds. INRIA, Rocquencourt, pp. 29–54.

38. Möncke, U. Grammar Flow Analysis. ESPRIT PROSPECTRA Project report S.1.3.-R-2.2, Univ. des Saarlandes, Saarbrücken, Mar. 1986, revised Jan. 1987. To appear in *ACM Trans. Progr. Languages and Systems.*

39. Parigot, D. Un système interactif de trace des circularités dans une grammaire attribuée et optimisation du test de circularité. Rapport de DEA, Univ. de Paris-Sud, Orsay, Sept. 1985.

40. Parigot, D. Mise en œuvre des grammaires attribuées: transformation, évaluation incrémentale, optimisations. Thèse de 3ème cycle, Univ. de Paris-Sud, Orsay, Sept. 1987.

41. Reiss, S. P. Generation of Compiler Symbol Processing Mechanisms from Specifications. *ACM Trans. Progr. Languages and Systems 5*, 2 (1983), 127–163.

42. Reps, T. *Generating Language-based Environments.* MIT Press, Cambridge, MA, 1984.

43. Reps, T. and Teitelbaum, T. *The Synthesizer Generator Reference Manual* 3rd edition. Springer-Verlag, New York–Heidelberg–Berlin, 1989.

44. Reps, T. and Teitelbaum, T. *The Synthesizer Generator.* Springer-Verlag, New York–Heidelberg–Berlin, 1989.

45. Riis-Nielson, H. Computation Sequences: A Way to Characterize Subclasses of Attribute Grammars. *Acta Inform. 19* (1983), 255–268.

46. Sagiv, S., Edelstein, O., Francez, N. and Rodeh, M. Resolving Circularity in Attribute Grammars with Applications to Data Flow Analysis. In *16th ACM Symp. on Principles of Progr. Languages* (Austin, TX, Jan. 1989). ACM Press, New York, NY, pp. 36–48.

47. Tiemann, M. D. Removing Redundancy in Attribute Grammars. Manuscript, Parallel Processing Program, Microelectronic and Computer Technology Corp., Austin, TX, July 1987.

48. Uhl, J., Drossopoulos, S., Persch, G., Goos, G., Dausmann, M., Winterstein, G. and Kirchgäßner, W. *An Attributed Grammar for the Semantic Analysis of ADA.* Lect. Notes in Comp. Sci., vol. 139, Springer-Verlag, New York–Heidelberg–Berlin, 1982.

49. Yeh, D. On Incremental Evaluation of Ordered Attributed Grammars. *BIT 23* (1983), 308–320.

50. Yeh, D. and Kastens, U. Improvements of an Incremental Evaluation Algorithm for Ordered Attributed Grammars. *ACM SIGPLAN Notices 23*, 12 (Dec. 1988), 45–50.