User-Adaptable Prettyprinting

Günther Blaschek, Johannes Sametinger
University of Linz, Austria

User-Adaptable Prettyprinting

G. Blaschek, J. Sametinger
University of Linz, Austria

SUMMARY

This paper presents a prettyprinter for high-level languages that can be adapted to the personal preferences of an individual user or to particular project conventions. The customization of the prettyprinter is done by means of a user profile with a set of parameters. The available parameters have been chosen with respect to minimality of the user interface and reasonable flexibility. The paper includes a complete list of all parameters with examples. The prettyprinter is fairly portable; it consists of a language independent back end and a front end that is created by a compiler generator from a formal description of the language to be processed. Currently, a Modula-2 and a Pascal version of the prettyprinter are implemented.

KEY WORDS Modula-2; Pascal; Prettyprinting; Program Formatting; Parameterization.

Introduction

As soon as high level languages have emerged, there has been a discussion on how to format programs. Nearly everybody admits that programs have to be well formatted in order not to obscure the logic of algorithms and data structures. But as a matter of fact nearly everyone prefers a different layout philosophy. Some examples can be seen explicitly in papers on programming style[1-11], and implicitly in an enormous number of books concerning high level programming languages[12]. Many papers have also been published on how to indent programs automatically[13-17].

Prettyprinting programs are useful to force a uniform indentation scheme in a software development project. They are still more important to improve maintainability of programs. Usually programs are not maintained by their authors; besides everybody is able to understand an unknown program easier if it is printed (indented) the way he prefers it. (With respect to prettyprinting, we disregard modularization and structured programming, which are actually more important than just a nice indentation). If a prettyprinter is flexible enough, programs can be indented in the maintainer's favorite style. And if one day someone else is responsible for the maintenance of this program, its layout can be modified easily.

Even though many papers on the subject of prettyprinting have already been published, hardly any work has yet dealt with a convenient parameterization of automatic formatters. Programs have been implemented and published using cryptical character codes describing how to indent programs. Occasional users of such programs can not be expected to change such descriptions in order to get their favorite result.

We designed a prettyprinter that enables everyone to define his preferred indentation style. This prettyprinter is implemented for different programming languages on different computers. With this paper, we hope to fill a gap dealing with the needs of programmers using prettyprinters.

In the following sections, we offer some general considerations and then point out the differences and equalities of indentation schemes. Then we present the user interface of our prettyprinter showing all the possibilities to define a specific layout. Finally, we briefly describe the implementation of the prettyprinter and discuss the results.

General Considerations

Strategies

The general strategy of prettyprinting is to scan the program symbol by symbol. Upon detection of special symbols in the source text (e.g. keywords), appropriate formatting actions are performed (e.g. starting a new line or indenting). This strategy has the advantage that the program to be formatted is not required to be syntactically correct. As an additional benefit, particular syntax errors can easily be found by formatting the program (especially missing ENDs).

Another approach is to perform a complete syntactic analysis on the program. With this strategy better results can be achieved, as more information is gathered (i.e. expressions can be structured into terms and factors, according to the operator hierarchy). Beside a complete lexical and syntactical analysis, this method also requires error recovery in order to deal with programs containing errors.

Flexibility

To change the layout of the prettyprinter output, the user either has to modify the source of the prettyprinter itself and recompile it[14], or he must change a formal description of the layout rules[11,13,18]. *Condict* et al.[19] suggest to insert directives into the program (as comments) to dynamically set formatting parameters. In this way, the prettyprinter can be forced to collect more than one statement per line or temporarily change the indentation depth (e.g. if part of the code is deeply nested). Although this approach seems to be very useful, the user has to modify his program if he dislikes the layout and wants to change it.

We do believe that flexibility is an important criterion of prettyprinters, because so many different layouts are used and preferred. It is one of the primary goals of this paper to present a convenient solution for this problem.

Comments

One of the most difficult problems in formatting programs is the adequate treatment of comments. Comments can be placed anywhere in a program and except for human readers it is hardly possible to determine where they logically belong.

Jackel[14] suggests placing comments on statements before the closing semicolon to enforce their correct placement by the formatting program. Another more general approach is made by *Fritzson*[13], who distinguishes three kinds of comments, one being placed on the left, one in the middle and one on the right side of a line. The distinction depends on the first character of the comment. *Mohilner*[20] proposes certain classes of comments that should be considered part of the standard format of the source text. *Rubin* uses left and right comment movement delimiters. These are "sets of terminals from the grammar that the prettyprinter cannot move comments to the left or right of, respectively"[18]. *Rose*[21,22] distinguishes so called pre- and post-comments. The distinction is made in a meta-syntax where any nonterminal can be pre- or postcommented and word-oriented comments are eventually folded automatically.

A First Approach

If a user wants a prettyprinter to indent a program the way he prefers it, he has to "tell" the prettyprinter his favorite indenting style. This should be possible in a comfortable way without the need to modify the program. If two programmers are talking about their favorite indentation rules, they presumably inspect sample programs of each other. So why not let the prettyprinter inspect a sample program, thus telling it "I want all my programs to be formatted like this one". Naturally it is too difficult for a program to analyze an arbitrary program text for indenting rules. Therefore let's take a small program the prettyprinter "knows". It is then possible to extract parameters (like indentation depth, number of blank lines between procedures, etc.) from this sample.

We implemented a prototype for such a prettyprinter that was able to format PL/M programs. We did not use a syntactically correct PL/M program as sample program, but rather a collection of typical language elements. For example, the desired layout of an IF statement could be described as follows:

```
IF
   expression
THEN
  statement
```

This example means that the expression of an IF statement should be indented three blanks, if it extends over more than one line (it does not mean that the expression should start on a new line). The keyword THEN should be aligned right under the IF and the statements of the THEN part should be indented two blanks.

This way of parameterizing is very useful for specifying different indentation depths for different language constructs. Of course, the expression starting on a new line might confuse occasional users. Aside from this, there are some more problems with this kind of parameterization:

- The user can change the layout of the sample program to be absolutely unformatted.
- It is easy to define indentation depths for various language elements, but it is rather difficult to specify under which circumstances a new line should be started or where spaces are to be inserted within, say, an expression.
- It is hardly possible to describe how comments should be treated. (We could not find a convenient solution.)

We found out that users were not happy with this kind of parameterization because they often were not able to describe their favorite style. Besides, we had to realize that users had many wishes we had not considered until then. Thus, before we made our next attempt, we tried to investigate what users really want.

Differences in Well Indented Programs

Most programmers prefer different indentations in some way. Even the authors disagree on some details. To learn more about these differences, we not only studied papers published on this subject but also wrote a little sample program in Modula-2 containing important language constructs, e.g.

- different declarations and statements
- very short and very long statements
- procedures with very long parameter lists

We asked various programmers to change the layout of this program according to their favorite style and inspected the results for equalities and differences. Some interesting facts are listed below.

- Most programmers use the same indentation depth over the whole program.

- Tabulators are used frequently in declarations and assignments, for example:

```
VAR
  i,j:       CARDINAL;
  tpart,ep:  INTEGER;
...
  i       := 1;
  longvar := anything;
```

- Some programmers use additional blanks very sparingly while others make frequent use of them.
- Blank lines are mainly used between procedures and between declaration and statement parts, but seldom to separate groups of statements.
- Sequences of short statements are often grouped together on a single line if they are tightly connected.
- The most different layouts can be observed with the IF statement. Programmers often prefer different styles depending on whether it contains an else part or elsif parts.
- Almost all comments immediately follow the program text they are intended to explain, i.e. after a single declaration or statement or after a procedure or module header. Only in a few cases they are used as headlines for a group of statements.

Many programmers try different layouts when confronted with rare constructs (e.g. very long parameter lists). Finally they arbitrarily decide which one to use.

Some indentation rules preferred by programmers cannot be automated at all, because they depend on the semantics of the program text (grouping of statements belonging together in a logical sense) or would need arbitrary look ahead (alignment of the ':=' signs in assignments).

Those things in mind, we made a new attempt to design our prettyprinter.

User Interface

Increasingly, parameter files ("user profile") are used to configure software tools. For example, the following parameters are used to control the editor's behavior and appearance on the personal computer Lilith[23]:

```
AutoIndent  enabled
Font        GACHA14
```

It is very easy to change these parameters (with a simple text editor) and to influence the behavior and user interface of several software tools available on the Lilith. As this concept is very easy to implement, understand, and use, we decided to adopt it for the prettyprinter, too.

In the rest of this section we describe the effects of the available parameters using examples (the range of the allowed values is written in parentheses). These parameters are specific to Modula-2, but most of them also apply to Pascal.

Line Width

Prettyprinters can be used for different purposes. In most cases, we will want to format an existing program in order to comply with some standard. The prettyprinted result is processed with a text editor again, where the desired layout has to be maintained manually (at least until the next prettyprinter run). For regular use, we will want our programs to be formatted with a maximal line width of, say, 80 characters (or whatever is best for the editor). For documentation purposes, we may want to print a listing on fanfold paper with up to 132 characters. In preparation for an article that is to be published in two columns, we might want to restrict the line width to 40 or 50 characters. The LineWidth parameter can be used to specify the number of characters that must not be exceeded by any symbol in the resulting output.

`LineWidth`	(40-132)	maximal number of characters per line

Indentation

The same indentation depth is applied over the whole program. We limit the maximum depth to four, as we believe that more blanks do not enhance the readability of a program[10], though others disagree[8].

`Indentation`	(1-4)	indentation depth used over the whole program

Blank Lines

Several parts of a program can be separated by one or more blank lines. The user can specify his choice with the following parameters:

`ImportSpace`	(0-2)	number of blank lines between import statements.
`DeclarationSpace`	(0-2)	number of blank lines between declaration parts (i.e. constant, type and variable declarations).
`ProcedureSpace`	(1-4)	number of blank lines between procedure declarations.

Due to the modularization concept of Modula-2, nested procedures and declarations are not as frequent as might be in programs written in languages like Pascal that do not support separate compilation. So we decided that only declarations and procedures at the outermost level are separated by blank lines while nested ones are not.

Blank Characters

It would certainly be inconvenient to list all possible locations in programs where blanks could be inserted. However, we could see in the inspected programs that some programmers used blanks very frequently while others did not. So we decided to offer only two different layouts:

`SeparatingBlanks`	(yes/no)	Use blanks to separate the symbols within statements?

Examples:

SeparatingBlanks = no

```
listSize:=(lastElem+1)*elemSize;
WriteCard(x,5);
```

SeparatingBlanks = yes

```
listSize := (lastElem + 1) * elemSize;
WriteCard (x, 5);
```

Program Density

As it is not possible to group statements together depending on their semantics, our prettyprinter just offers the facility to collect structured statements if they fit on a single line entirely.

`PackCompounds`	(yes/no)	collect short statement sequences in a single line?

Examples:

PackCompounds = yes

```
IF a < b THEN c := a; a := b; b := c END
```

PackCompounds = no

```
IF a < b THEN
  c := a;
  a := b;
  b := c
  END
```

Declarations

As mentioned above, constant, type and variable declarations can be separated by blank lines. Within the groups, every single declaration starts on a new line. We found out that tabulators in declarations can enhance their readability considerably.

`DeclarationTab`	(0-30)	column number for right sides of declarations; if the value 0 is specified, no tabulators are used.

Examples:

DeclarationTab = 0

```
CONST
  bufsize = 100;
  eol = 4C;
VAR
  ch: CHAR;
  buffer: ARRAY [0..bufsize-1] OF CHAR;
```

DeclarationTab = 20

```
CONST
  bufsize =         100;
  eol =             4C;
VAR
  ch:               CHAR;
  buffer:           ARRAY [0..bufsize-1] OF CHAR;
```

These tabulators are used in nested declarations, too. As another feature, we provide the possibility of indenting declarations as a whole.

`DeclarationIndent`	(yes/no)	indent declaration parts?

Examples:

DeclarationIndent = no

```
PROCEDURE DoAnything;
CONST
  eol = 4C;
PROCEDURE Internal;
VAR
  x: INTEGER;
BEGIN
  (*Body of Internal*)
  END Internal;
BEGIN
  (*Body of DoAnything*)
  END DoAnything;
```

DeclarationIndent = yes

```
PROCEDURE DoAnything;
  CONST
    eol = 4C;
  PROCEDURE Internal;
    VAR
      x: INTEGER;
    BEGIN
      (*Body of Internal*)
      END Internal;
  BEGIN
    (*Body of DoAnything*)
    END DoAnything;
```

As can be seen in this example, also nested declarations (and procedures) are indented. The indentation depth depends on the value specified for the *Indentation* parameter.

If Statements

We distinguish between three different indentation philosophies concerning if statements. The prettyprinter does not make any differences depending on whether an else part and/or elsif parts are present. The choice is specified by the following two parameters.

`NewLineThen`	(yes/no)	shall the keyword THEN start on a new line?
`IndentThen`	(yes/no)	shall the keyword THEN be indented? (only applicable, if NewLineThen=yes has been specified).

Depending on the settings of these two parameters, three different indentation schemes can be achieved, as shown in the following examples.

NewLineThen = no

```
IF ch = '=' THEN
  typ := leqsy; NextCh
ELSIF ch = '>' THEN
  typ := neqsy; NextCh
ELSE
  typ := lthsy
END;
```

NewLineThen = yes; IndentThen = no

```
IF ch = '='
THEN
  typ := leqsy; NextCh
ELSIF ch = '>'
THEN
  typ := neqsy; NextCh
ELSE
  typ := lthsy
END;
```

NewLineThen = yes; IndentThen = yes

```
IF ch = '='
  THEN
    typ := leqsy; NextCh
ELSIF ch = '>'
  THEN
    typ := neqsy; NextCh
  ELSE
    typ := lthsy
  END;
```

Case Statements

The user can choose one of three different layouts similar to the if statement.

`NewLineCase`	(yes/no)	shall the '|'-sign appear on a new line?
`IndentCaseLabs`	(yes/no)	shall the case labels be indented? (only applicable, if NewLineCase=no has been specified)

Examples[12]:

NewLineCase = yes

```
CASE state OF
  single: WriteString('single')
| married:
    WriteString('married');
    WriteCard(spouseId,10);
    WriteCard(NoOfChildren,4);
    WriteDate(wedding)
| widowed:
    WriteString('widowed');
    WriteDate(death)
END;
```

NewLineCase = no; IndentCaseLabs = no

```
CASE state OF
single: WriteString('single') |
married:
  WriteString('married');
  WriteCard(spouseId,10);
  WriteCard(NoOfChildren,4);
  WriteDate(wedding)|
widowed:
  WriteString('widowed');
  WriteDate(death)
END;
```

NewLineCase = no; IndentCaseLabs = yes

```
CASE state OF
  single: WriteString('single') |
  married:
    WriteString('married');
    WriteCard(spouseId,10);
    WriteCard(NoOfChildren,4);
    WriteDate(wedding) |
  widowed:
    WriteString('widowed');
    WriteDate(death)
END;
```

The Keyword END

In Modula-2, all structured statements (except repeat/until) are terminated by the keyword END. As programmers disagree (like the authors do), whether or not to indent the keyword END, both possibilities are offered.

`IndentEnd`	(yes/no)	indent the keyword END?

Examples:

IndentEnd = yes

```
FOR i:=1 TO n DO
  IF a[i]#b[i] THEN
    RETURN FALSE
    END
  END;
RETURN TRUE;
```

IndentEnd = no

```
FOR i:=1 TO n DO
  IF a[i]#b[i] THEN
    RETURN FALSE
  END
END;
RETURN TRUE;
```

Comments

Many prettyprinters have problems with the treatment of comments, because it is difficult to determine whether they belong to the preceding or to the subsequent program text. We decided for the post-commented manner, i.e. all comments are treated as if they belonged to the preceding program text.

Examples:

```
CONST
  bufsize = 100; (*size of input buffer*)
  eol = 4C; (*end of line*)
BEGIN
  ...
  GetSym; (*read the first input symbol*)
```

Long comments (that contain at least one end-of-line) always start on a new line. A particular problem is that the structure of such comments must not be destroyed. For example, the statement

```
GetSym; (* read the first input symbol
          (this assigns values to sym and attr) *)
```

should not be formatted as

```
GetSym;
(* read the first input symbol
          (this assigns values to sym and attr) *)
```

but rather as

```
GetSym;
(* read the first input symbol
  (this assigns values to sym and attr) *)
```

Comments in Declarations

If programmers use comments in declarations, they often align the left sides in the same way as, for example, type declarations. So we enable users to specify a column where comments in declarations should start.

`DclCommentTab`	(0-50)	position of comments in declarations.

As with the parameter *DeclarationTab*, the value 0 means that no tabulators are to be used.

Examples:

DclCommentTab = 0

```
CONST
  bufsize = 100; (*size of input buffer*)
  eol = 4C; (*end of line*)
VAR
  ch: CHAR; (*current input character*)
  buffer: ARRAY [0..bufsize-1] OF CHAR; (*input buffer*)
```

DclCommentTab = 30

```
CONST
  bufsize = 100;              (*size of input buffer*)
  eol = 4C;                   (*end of line*)
VAR
  ch: CHAR;                   (*current input character*)
  buffer: ARRAY [0..bufsize-1] OF CHAR; (*input buffer*)
```

Note that the comment `(*input buffer*)` is appended with just one leading blank, since the line is already longer then 30 characters.

We recommend using both DeclarationTab and DclCommentTab, for example:

DeclarationTab = 15; DclCommentTab = 30

```
CONST
  bufsize =    100;           (*size of input buffer*)
  eol =        4C;            (*end of line*)
VAR
  ch:          CHAR;          (*current input character*)
  buffer:      ARRAY [0..bufsize-1] OF CHAR; (*input buffer*)
```

Comments after the Keyword END

In Modula-2 the keyword END terminates several compound statements without indicating the kind of the statement being terminated. In order to increase the readability of deeply nested statements, the prettyprinter can be forced to append auxiliary comments after each END automatically.

`EndComments`	(yes/no)	automatic END comments desired?

Example:

EndComments = yes

```
FOR i:=1 TO n DO
  IF a[i]#b[i] THEN
    RETURN FALSE
  END
END (*FOR*);
RETURN TRUE;
```

Note that these comments are suppressed at the innermost level, since it is obvious in these cases, where the END belongs (in particular when these statements are very short). Comments are also suppressed if the input text already contained a (possibly more expressive) comment following the keyword END.

We end this section by listing all parameters together with their default values:

```
LineWidth          78
Indentation        2
SeparatingBlanks   no
PackCompounds      yes
DeclarationIndent  no
DeclarationTab     25
DclCommentTab      45
ImportSpace        0
DeclarationSpace   1
ProcedureSpace     2
IndentEnd          no
EndComments        yes
NewLineThen        no
IndentThen         no
NewLineCase        no
IndentCaseLabs     yes
```

These are the parameters we use for Modula-2. For other languages some additional parameters might be useful while others will not apply. Our Pascal prettyprinter, for example, provides two additional parameters:

`upperKeys`	(yes/no)	convert keywords to capital letters?
`lowerKeys`	(yes/no)	convert keywords to lower case letters?

If both upperKeys and lowerKeys are set to "no", all keywords are left unchanged.

Implementation

Language Independent Prettyprinting Algorithm

For the implementation of the formatting program we used a language independent prettyprinting algorithm by D. C. Oppen[17]. This algorithm receives the lexical tokens of a program with embedded control information.

Parts of a program that logically belong together are grouped with the following control characters:

{ ... start of a block
} ... end of a block

Consider the following example:

```
{ { IF { x<y } THEN }
{ y:=y-x }
{ ELSE { x:=x-y } }
END }
```

The algorithm always tries to place a logical block on a single line, i.e.

```
IF x<y THEN y:=y-x ELSE x:=x-y END
```

If the block is too long, then the subblocks are either placed each on a single line (consistent line breaking) or on the same line as long as possible (inconsistent line breaking). The result with insufficient line length and consistent line breaking would be

```
IF x<y THEN
  y:=y-x
ELSE x:=x-y
END
```

Lexical tokens are treated as elementary subblocks. Inconsistent line breaking would lead to one of the following results (depending on the line length):

```
IF x<y THEN y:=y-x ELSE x:=x-y
END
```

or

```
IF x<y THEN y:=y-x
ELSE x:=x-y END
```

In the example

```
{ IF { { (x<a) AND } { (x<b) AND } { (x<c) AND } { (x<d) AND }
{ (x<e) AND } { (x<f) AND } { (x<g) } } THEN }
```

a line break after IF would occur because the following subblock does not fit into the current line. This would result in

```
IF
   (x<a) AND (x<b) AND (x<c) AND (x<d) AND
   (x<e) AND (x<f) AND (x<g) THEN
```

As this is not desired, positions where line breaks may occur are marked explicitly, i.e. the following control character is inserted:

brk ... possible line break

If a blank is inserted after IF instead of a break, the following result is possible:

```
IF (x<a) AND (x<b) AND (x<c) AND (x<d) AND
   (x<e) AND (x<f) AND (x<g) THEN
```

For further control of the layout, some additional information is passed to the algorithm:

- indentation depth of subblocks
- number of blanks between subblocks

The ideas and the implementation of this algorithm are described in detail by Oppen.

Modifications to the Algorithm

Since the algorithm by D. C. Oppen does not support tabulators, we had to attach slight modifications, i.e. add new control symbols:

tab ... specifies the position where the next token should start

A tab symbol is inserted before each constant or type specification.
To avoid the following output

```
VAR
  addr,i,j,lactblocklist,length,stackp,val1,
  val2:                CARDINAL;
  shortexpr:           BOOLEAN;
```

we added two more symbols, namely

setMargin ... set right margin to a specified value
endMargin ... reset right margin to the previous value

These symbols can be used to temporarily set the right margin (that must not be exceeded) to a column left of the relevant tabulator position. This results in the following output:

```
VAR
  addr,i,j,
  lactblocklist,
  length,stackp,
  val1,val2:          CARDINAL;
  shortexpr:          BOOLEAN;
```

Obviously, this image is more readable and appealing than the previous one.

The symbols *tab*, *setMargin* and *endMargin* are implemented quite similar to the ones already described by Oppen.

Generating the desired output

The program to be prettyprinted must be analyzed syntactically. Common compiler writing techniques can be used to decompose it into its symbols and to insert the control information into the token stream. We used two different techniques for this purpose.

Our prettyprinter for Modula-2 receives the tokens from a syntax tree that was previously created by a compiler. (The Modula-2 prettyprinter is actually part of a software development tool[24] rather than a separate tool.) The token stream for the prettyprinter is generated by traversing the tree recursively.

The Pascal implementation is a stand-alone tool. It was generated by means of a compiler compiler[25] which takes an attributed grammar as input and generates a table driven LL(1) parser. The following rule for a WHILE statement shall serve as an example, how the "translation" process is described in the attributed grammar (semantic actions are Pascal statements; they are enclosed by the keywords *sem* and *endsem*):

```
WhileStat =
    WHILE             sem      IF NOT firststat THEN Break(1,0);
                               BeginBlock(consistent);
                               BeginBlock(inconsistent);
                               WriteTerminal;
                               PrintChar(' ')
                      endsem
    Expression
    DO                sem      Break(1,0); WriteTerminal;
                               EndBlock; Break(1,ind);
                               firststat := TRUE
                      endsem
    Statement         sem      EndBlock; firststat := FALSE
                      endsem
```

The procedures *Break*, *BeginBlock* and *EndBlock* are used for transmitting the symbols *brk*, '{' and '}', respectively. The global variable *firststat* is used to determine whether or not a statement is the first within a statement sequence. The value *ind* specifies the indentation depth; it is taken from the user profile.

To avoid passing tokens from the scanner to the parser, and then again to the prettyprinter, the scanner already puts the symbols read into the token stream. This is not only convenient and efficient, but also mandatory, since the parser does not know anything about comments, which must be preserved in the prettyprinted program. However, the parser sometimes needs to insert additional control information *before* the token just recognized by the scanner. In the example above, a consistent and an inconsistent break must be passed to the prettyprinter before the keyword WHILE. This problem is solved by delaying the transmission of program tokens, i.e. the scanner emits a token, when the parser requests the next one. The semantic procedure *WriteTerminal* can be used to force the scanner to emit the actual token. This is necessary in some situations, where control information has to be inserted *after* the actual token.

The symbol sequence generated by the rule above is as follows:

```
brk { { WHILE ' ' Expression brk DO } brk Statement }
```

The outer block forces consistent line breaking, i.e. the statement starts on a new line if the actual statement sequence does not fit on a single line. Inconsistent line breaking is used by the inner block because otherwise very long expressions would be splitted over too many lines. The proper layout of the expression is guaranteed by the rule for *Expression*.

Conclusions

The presented prettyprinter yields pleasing results. Applying it consequently, we got used to a uniform indentation scheme over all our programs. The small deviations from our personal style do not bother us any longer.

Parameterization

The chosen user interface turned out to be quite a success. This means that other programmers adopted this kind of parameterization, too. One of the advantages of this method is that new parameters can easily be added (if it turns out that more of them should be offered). One disadvantage is that the meaning of some parameters might not be understandable without an explaining text. (However, short explanations are sufficient in most cases).

Implementation

The use of an attributed grammar (together with a parser generator) and a language independent prettyprinting algorithm enhances the portability of the formatting program. Prettyprinters for other languages can easily be implemented just by writing a new attributed grammar and a scanner for the new language. The use of a parser generator has yet another advantage because the generated table driven parser provides automatic error recovery.

Benefits

The following are the most important advantages of our prettyprinter together with the parameterization method:

- It produces compact but still clear (i.e. highly readable) programs.
- Tabulators in declarations enhance readability significantly.
- Insertion of auxiliary comments after the ENDs eases the reading of deeply nested programs.
- The result is always well indented no matter what parameter values are specified.
- The prettyprinter is easy to use and easy to adapt to personal preferences.
- The use of a compiler generator for the front end and a language independent algorithm as back end results in a highly portable tool.

We believe that our current implementations represent acceptable compromises between efficiency of the prettyprinter and good results of the formatted programs.

Deficiencies

The most severe disadvantage of the system is that it is not applicable to programs containing syntax errors. Another disadvantage is the low speed of the prettyprinter. The main reason for this is that the prettyprinter has to analyze the source text completely. But of course this is necessary to arrive at an acceptable output. Yet the programs are not formatted perfectly (which probably can never be done automatically). Some points we dislike are:

- The position of the tabulators in declarations should depend on the extent of the declarations rather than on fixed values specified by the user.
- Tabulators would be nice within record fields, too.
- Sometimes inner declarations are very extensive and then should be divided by blank lines as is done with declarations at the outermost level.
- Although our approach for treating comments yields satisfactory results in most cases (especially when writing programs with post-processing by the prettyprinter in mind - what we are now doing), further heuristics would be necessary for better results.
- It is not possible to align the ':=' signs or the right sides of assignment statements, as in:

```
WITH stmt^ DO
  position  := pos;
  decLabel  := label;
  next      := NIL;
  type      := NIL;
  stmtScope := currStmtScope;
END;
```

Perhaps some of the deficiencies mentioned above will be eliminated in future work, but one always has to compromise. Our goal was to provide a maximum of convenience and adaptability with a minimal set of parameters.

References

1. Bailes P.A., Salvadori A.: "A Semantically-based Formatting Discipline for Pascal" *Software - Practice and Experience*, Vol. 14, No. 3, 235–251 (1984).

2. Baldwin R.R.: "Systematic Indentation in PL/I: Minimizing the Reduction in Horizontal Space" *ACM Sigplan Notices*, Vol. 21, No. 9, 22–26 (1986).

3. Hueras J.F., Ledgard H.F.: "An automatic formatting program for Pascal" *ACM Sigplan Notices*, Vol 12, No. 7, 82–84 (1977).

4. Peterson J.L.: "On the Formatting of Pascal Programs" *ACM Sigplan Notices*, Vol. 12, No. 12, 83–86 (1977).

5. Sale A.H.J.: "Stylistics in Languages with Compound Statements" *Australian Computer Journal*, Vol. 10, No. 2, 58–59 (1978).

6. Yehudai A.: "Automatic Indentation Versus Program Formatting" *ACM Sigplan Notices*, Vol. 15, No. 10, 85–87 (1980).

7. Heckert R.: "A Pascal Indentation Philosophy" *Computer Language*, 37–39 (1985).

8. Cooper B., Gallagher B., Sekercan G.: Feedbacks to "A Pascal Indentation Philosophy"7 *Computer Language*, 7 (1985).

9. Leavens G.T.: "Prettyprinting Styles for Various Languages" *ACM Sigplan Notices*, Vol. 19, No. 2, 75–79 (1984).

10. Miara R.J. et al.: "Program Indentation and Comprehensibility" *Communications of the ACM*, Vol. 26, No. 11, 861–867 (1983).

11. Woodman M.: "Formatted Syntaxes and Modula-2" *Software – Practice and Experience*, Vol. 16, No. 7, 605–626 (1986).

12. Wirth N.: *Programming in Modula-2* (3rd edition); Springer (1985).

13. Fritzson P.: *Adaptive Prettyprinting of Abstract Syntax applied to Ada and Pascal*; Research Report, University of Linköping/Sweden (1983).

14. Jackel M.: "A formatting parser for Pascal programs" *ACM Sigplan Notices*, Vol. 15, No. (7&8), 58–63 (1980).

15. Mateti P.: "A Specification Scheme for Indenting Programs" *Software – Practice and Experience*, Vol. 13, 163–179 (1983).

16. Mateti P., Jaffar J.: "A Correctness Proof of an Indenting Program" *Software – Practice and Experience*, Vol. 13, 199–226 (1983).

17. Oppen D.C.: "Prettyprinting" *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 4, 465–483 (1980).

18. Rubin L.F.: "Syntax-Directed Pretty-Printing – A First Step Towards a Syntax-Directed Editor" *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 2 (1983).

19. Condict N.M., Marcus R.L., Mickel A.: *Spruce, a Pascal Program Formatter*; University Computer Center, University of Minnesota, Included in Pascal 6000 Release 3.

20. Mohilner P.R.: "Prettyprinting Pascal Programs" *ACM Sigplan Notices*, Vol. 13, No. 7, 34–40 (1978).

21. Rose G.A.,Welsh J.: "Formatted Programming Languages" *Software – Practice and Experience*, Vol. 11, No. 12, 651–669 (1979).

22. Rose G., Roper T.: "Generation of Program-Preparation Systems for Formatted Languages" *Proceedings IFIP Paris* (1983).

23. Wirth N.: *The Personal Computer Lilith*; ETH Report #40, Institute of Computer Science, ETH Zürich (1981).

24. Blaschek G., Pomberger G.: "Moses – A Graphics Oriented Software Development Environment" *Proceedings ACM Computer Science Conference* (1987).

25. Rechenberg P., Mössenböck H.: *Ein Compiler-Generator für Mikrocomputer*; Hanser - Verlag (1985). (in German; to appear in English in 1988).