PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 35, NO. 2, PP. 125–146 (1991)

AN ENVIRONMENT FOR ENGINEERING EXTENDED AFFIX GRAMMAR ENVIRONMENTS[1]

M. G. J. VAN DEN BRAND

Department of Informatics
Faculty of Mathematics and Informatics
University of Nijmegen, The Netherlands

Received: Sept. 5, 1990.

Abstract

Existing formalisms for the specification of programming environments are complex and strongly biased by the problems of environment generation. It has been investigated whether a simple two-level grammar, describing a programming language, can be used without further modification for the generation of an environment for that language.

We believe that there is enough information in most language definitions — albeit implicitly — to generate most of the tools used in syntax-directed editors.

This paper proposes some simple and elegant improvements in the use of placeholders and templates, and in the unparsing mechanism. Although the improvements are implemented in a completely newly designed prototype they can also be applied to existing syntax-directed editors to improve their workability.

Keywords: programming environments, extended affix grammars.

1 Introduction

Currently, many experienced as well as inexperienced programmers are working with programming environments, since such environments significantly simplify the process of writing programs. The environments are either hand-made or generated from some specification formalism. Such a formalism is usually based on attribute grammars. Nevertheless constructing an environment is still a major effort because each part of the environment must be explicitly specified.

A programming environment generator is a useful tool for a language designer as well, to check whether the language under development is easy to use. If, however, the generation of an environment requires a lot of extra effort the language designer may refrain from prototyping.

We are, therefore, investigating the possibility of using a given EAG (Extended Affix Grammar, WATT, 1974), describing both the syntax and

[1]This project is financed by NWO, the Dutch Organisation for Scientific Research; project number: 612–317–020

static semantics of a language, as a specification of a programming environment for that language. It is our ambition to safeguard the EAG writer from the need to adapt his language definition. This research is particularly characterized by the fact that the EAG specification need not be changed to derive the different parts of the programming environment.

We have generated a syntax-directed editor for the EAG formalism, given in appendix A, to test the prototype of the generator and to explore the functionality of the generated editors.

2 Syntax-directed Editor Generators

We have to consider three aspects of a system for generating programming environments.

- The generator itself, which is not very interesting.

  It is merely a transducer from the formalism to the environment. A transducer is sufficient if the specification contains explicit definitions for all parts of the environment, which is usually the case. In several existing systems, such as the Synthesizer Generator (REPS, 1984; REPS and TEITELBAUM, 1989a), YACC (JOHNSON, 1975) is used for the generator.

- The input, namely the formalism to specify the language for which the environment is generated.

  Various specification formalisms have been proposed. They must be expressive, since they have to be read and written by human beings, and they must allow for an automatic implementation. Most of the specification formalisms for generating programming environments are based on attribute grammars (KNUTH, 1968), usually enriched with unparsing rules, tools for the definition of templates and a mechanism for the specification of the abstract syntax of the language. As a result, most of the resulting specifications consist of several parts. HEERING (1983) describes three major parts: the syntax rules, the semantic rules, and the unparsing rules. KLINT (1983) further distinguishes lexical, concrete, and abstract syntax rules and static and dynamic semantic rules. In attribute grammars, syntax and static semantics are defined by different means. The static semantics are usually operationalized using a programming language such as C or Pascal.

  A second way of specifying an environment is by means of an algebraic specification (BERGSTRA et al, 1989). The specification formalism ASF/SDF of GIPE (HEERING et al, 1986) is based on algebraic specifications. The syntax and the static and dynamic semantics are defined in the same formalism.

- The output, namely the generated environment.

  A complete programming environment should consist of several parts: certainly not only a syntax-directed editor and an unparser but also an interpreter and a compiler, and possibly even a debugger.

  There are two fundamentally different classes of syntax-directed editors: template editors and text editors – hybrid editors combine features of both editor classes.

  Template editors. The editor generated by the ALOE generator of the GANDALF system (MEDINA-MORA and FEILER, 1981; MEDINA-MORA, 1982) is a template editor. The greatest drawback of such an editor is the tedious way of editing small-scale constructs such as expressions. It is also often not easy to replace a construct by another construct, for example a while-loop by an until-loop. On the other hand it is impossible to create syntactically incorrect programs with this kind of editor.

  Text editors. The editor used in the environment of the SAGA system (CAMPBELL and KIRSLIS, 1984) is a text editor. The disadvantage of such an editor is that the syntax-directedness is limited to the way in which the structure of the abstract syntax tree is shown to the user.

  Hybrid editors. The editors in the environments generated by the Synthesizer Generator and the PSG system (BAHLKE and SNELTING, 1986) are hybrid editors. A hybrid editor offers the possibility of using both template editing and text editing. The disadvantages of the other two types of editors are eliminated, whereas their advantages are combined.

3 Simple Specification Formalism

Most designers of environment generators are developing more and more complicated specification formalisms to specify the various parts of the environments. We believe that most of the relevant information for the generation of these parts can be derived implicitly. Our main objective is to keep the formalism as simple as possible.

The basis of our formalism is Extended Affix Grammar, which initially was developed for the specification of compilers (MEIJER, 1986). An EAG can be characterized as a context-free grammar decorated with attributes; EAG and attribute grammars are to a certain extent very similar (MEIJER, 1990).

In this paper we concentrate on the syntactical part of the generated editors, therefore only the context-free part of the EAG will be considered in the rest of this paper.

The specification formalism may be 'polluted' in three ways:

- adaptions;
- extensions;
- restrictions.

These three can be avoided if the specification formalism is elegant and the generators are powerful enough. We will not give an exhaustive list of unnecessary restrictions, extensions and adaptions but give a striking example of each.

3.1 Adaptions

The specification writer has to rewrite the grammar in some way before he can transform it into a specification. Possible rewritings are for example: left recursion elimination and the separation of lexical and syntactical properties of the grammar. This last rewriting will be further clarified.

Most systems known to us are based on YACC (JOHNSON, 1975) and LEX (LESK, 1975), each of these tools has its own specification formalism. In formalisms of systems based on these tools the YACC and LEX parts are clearly recognizable. The frequent use of these tools by most people made the YACC/LEX way of specifying grammars more or less a standard, although it is not the most natural way of writing context-free grammars. The Van Wijngaarden style (VAN WIJNGAARDEN et al, 1976), which is the basis of our formalism, frees the specification writer from the burden of transforming his intuitive ideas into a (YACC) program.

3.2 Extensions

The specification writer has to add text not directly related to the grammar in order to specify some part or tool in the editor. Examples are the specification of templates, names of the placeholders and unparsing rules.

Most syntax-directed editors allow the use of templates. In the specification formalism these templates are defined explicitly in some way. In the formalism SSL (REPS and TEITELBAUM, 1989b), for example, the templates for a nonterminal [exp] are defined as:

```
transform exp on "+" [exp]:  Sum([exp], [exp]),
              on "-" [exp]:  Diff([exp], [exp]),
              on "*" [exp]:  Prod([exp], [exp]),
              on "/" [exp]:  Quot([exp], [exp])
              ;
```

Where the unparsing rules for the nodes Sum, Diff, Prod, Quot are defined as:

```
exp:   Sum  [@ ::= "(" @ "+" @ ")"]
   |  Diff  [@ ::= "(" @ "-" @ ")"]
   |  Prod  [@ ::= "(" @ "*" @ ")"]
   |  Quot  [@ ::= "(" @ "/" @ ")"]
   ;
```

This kind of information is also implicitly available in the context-free grammar and need not be explicitly defined. This will reduce both the amount of work and the possibility of making mistakes.

3.3 Restrictions

The specification writer is not allowed to transform an arbitrary grammar into a specification because the incorporated parser in the generated syntax-directed editor may not be powerful enough.

Most generators are based on YACC, thus the underlying grammar of the specification has to be LALR(1). Although most programming languages fulfil this condition, it may be interesting to experiment with syntax-directed editing of ambiguous grammars.

We have solved this problem by using an elegant specimen of a backtrack parser (AHO and ULLMAN, 1972; KOSTER, 1975). The consequence is an increase in complexity, but because real syntax-directed editors are highly incremental this reduction of parsing speed is not observable in the performance of the system.

4 Generated Editors and Ambiguous Grammars

Imposing no restrictions on the grammar may lead to ambiguity. By using a backtrack parser we are able to edit the largest class of languages in a syntax-directed way. Before we describe the mechanism in more detail some remarks on the parser are necessary.

4.1 Backtrack Parser

For a description of backtrack parsing in general we refer to (AHO and ULLMAN, 1972) and for a description of our version to (KOSTER, 1975).

The backtrack parser described in (AHO and ULLMAN, 1972) stops after finding a successful parse, to find the other parses the parser must be forced in backtrack mode.

The syntax tree is also built by backtracking, i. e. in the recognition phase the tree is built and in the backtrack phase it is dismantled. The syntax tree is complete whenever a successful parse is found.

4.2 Our Solution

The edit actions have to be performed on the syntax tree, which is built by the parser. As the tree will be dismantled in the backtrack phase it has to be copied each time a successful parse is found. In case of an ambiguous input sentence the parser may build several different syntax trees. The editor can only use one tree and selecting one of the yielded trees would contradict our philosophy of generality.

This is solved by combining all syntax trees in one parse tree, which has a 3-dimensional structure. For this purpose, a special kind of node ambiguous is introduced. Its sons are subtrees which have the same root and represent the same substring in the input sentence. The internal structure of each of these subtrees is different.

In each two syntax trees yielded by the parser for the same input sentence the following condition holds; either two different alternatives were chosen for the start nonterminal or, in both trees, the same alternative was chosen and the condition holds recursively for each nonterminal in the right hand sides of the corresponding alternatives. In *Fig. 1* the two nodes for which different alternatives were chosen have the labels s_1 and s_2, respectively.

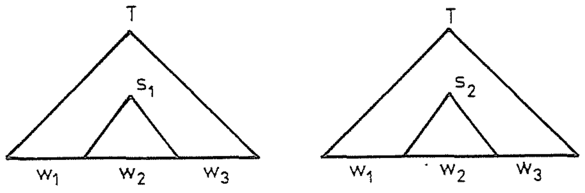

Fig. 1. Two arbitrary parse trees for an ambiguous sentence

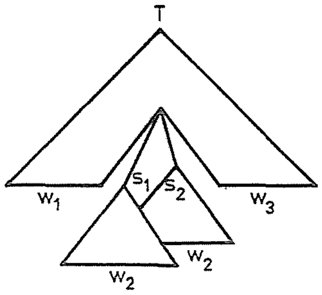

Fig. 2. 3-dimensional parse tree for an ambiguous sentence

The frontier of s_1 equals the frontier of s_2, however, they represent different alternatives of the same nonterminal in the grammar. In *Fig. 1* the two parse trees are combined in one 3-dimensional parse tree.

4.3 Unparsing of 3-dimensional Parse Trees

The unparser, to be discussed in Section 6, is in fact a tree-traversal routine. The pretty print of a program is obtained by inserting blanks and newlines between the terminals. The unparser knows how each type of node must be pretty printed.

For an ambiguous input sentence it is possible that the parses of this sentence have different pretty prints. If no unique pretty print can be found, not pretty printing the text at all seems to be the best alternative. In fact it is also a way of stressing which part of the program is ambiguous. So, the unparse routine does not insert layout in the subtrees of a node of type `ambiguous`.

4.4 Ambiguity and Incremental Techniques Contradict

In order to guarantee that the user of the environment generated for an ambiguous grammar gets all possible information, the entire program must be reparsed after each alteration. All facilities and techniques introduced to maximalize the amount of incremental work seem to be superfluous.

Although ambiguity in general renders incremental techniques useless, the techniques introduced above are still very useful in Section 5 where the extension of the language with untyped placeholders leads to very 'local' ambiguity.

5 Another Look at Placeholders and Templates

The tradional hybrid editor uses placeholders and templates. The derivation of templates from the specification formalism will be discussed in Section 5.2. In Section 5.1 we will improve the flexibility of the hybrid editor by introducing a new kind of placeholder.

5.1 Placeholders

In Section 2 we discussed the restrictions of several types of syntax-directed editors. The hybrid editor proved to be the most flexible. We believe, however, that this type of editor can be made even more flexible.

Editors generated by the Synthesizer Generator offer the possibility of editing the text of a complete syntactical construct using text edit mode. This selected construct may contain several placeholders. The parser is always called immediately after leaving the text mode. The changed program text is rejected if it contains placeholders, because the parser cannot recognize placeholders. So, the text containing placeholders created by the editor cannot be recognized by the same editor.

The inflexibility is caused by a too sharp distinction between text and template mode. It is not permitted to manipulate placeholders in text mode, other than replacing them by plain program text.

The traditional placeholder consisting of a special open bracket and close bracket enclosing the name of the replaced syntactic construct will be called a *typed* placeholder in the rest of this paper. We also introduce the *untyped* placeholder; a new kind of terminal symbol not associated with some specific syntactic structure.

The introduction of these two types of placeholders enable the user to edit syntactic structures containing several placeholders without replacing them all, as well as to replace placeholders by templates or by plain text. Furthermore, it will be possible to insert placeholders in the text mode of the hybrid editor.

5.1.1 Typed Placeholders

The parsers in the hybrid editors of other systems are not able to recognize typed placeholders. However, by a simple extension of the specification recognition becomes possible.

As an example we modify an SSL specification. In appendix A of (Reps and Teitelbaum, 1989b) a specification of a simple desk-calculator is given. We will give the new production rule for Exp in the Parse syntax part.

```
Exp  ::= ('[exp]')  {$$.abs = Null();}
   |  (INTEGER)     {$$.abs = Const(STRtoINT(INTEGER));}
   |  (Exp '+' Exp) {$$.abs = Sum(Exp$2.abs,Exp$3.abs);}
   |  (Exp '-' Exp) {$$.abs = Diff(Exp$2.abs,Exp$3.abs);}
   |  (Exp '*' Exp) {$$.abs = Prod(Exp$2.abs,Exp$3.abs);}
   |  (Exp '/' Exp) {$$.abs = Quot(Exp$2.abs,Exp$3.abs);}
   |  ('(' Exp ')') {$$.abs = Quot(Exp$2.abs;}
   ;
```

The parser in the new generated editor is now able to recognize typed placeholders in text mode. However, the specification writer has to do the adaption, which contradicts our philosophy.

Instead of letting the specification writer transform his specification our system implicitly transforms the specification. Each production rule in the context-free grammar is extended with an extra alternative to recognize the typed placeholder.

$$A : \ldots \quad . \Rightarrow \begin{cases} A: & [A]; \\ A: & \ldots \quad . \end{cases}$$

The member in the right hand side of the new alternative is not just a terminal symbol. The placeholder replaces a complete language construct, which should also be marked in the syntax tree.

5.1.2 Untyped Placeholders

The introduction of typed placeholders increases the flexibility of the editor, but to use these typed placeholder optimally, the user of the editor must know the exact names of all placeholders. This is impossible for a language such as Algol 68 (VAN WIJNGAARDEN et al, 1976).

Therefore, we allow the user to specify the place where he wants a placeholder by a special kind of placeholder symbol which represents almost all syntactical constructs in the language.

To recognize untyped placeholders the parser is extended in a way similar to that for typed placeholders. Each production rule gets an extra alternative to recognize the untyped placeholder symbol '□'.

$$A : \ldots . \Rightarrow \begin{cases} A: & \square; \\ A: & [A]; \\ A: & \ldots \quad . \end{cases}$$

This extension of the grammar causes ambiguity. Parsers in hybrid editors of other systems do not offer this facility, they do not allow the grammar to be ambiguous.

5.1.3 3-dimensional Syntax Trees

The main problem is the assignment of names to the untyped placeholders during recognition. An untyped placeholder can be replaced by several typed ones, each of the substitutions results in a successful parse and a corresponding syntax tree. These syntax trees can be combined using the technique discussed in Subsection 4.2.

Suppose we have generated a syntax-directed editor for the language specified in appendix A. Almost all production rules are extended with the extra alternatives. The user of the editor has inserted the text:

`[startnonterminal]□ □ : .`

The parser has found 8 different parses for the string. We give only the subtrees for the substring '□ □:.'. All these subtrees start with the nonterminal `productionrules`. The parser yields among others the parse trees shown in *Figs 3* and *4*.

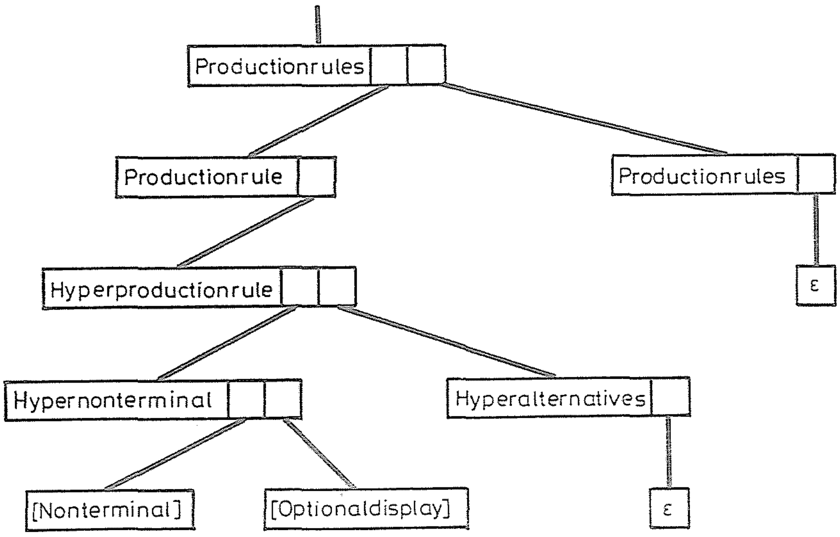

Fig. 3. Parse tree of the first parse

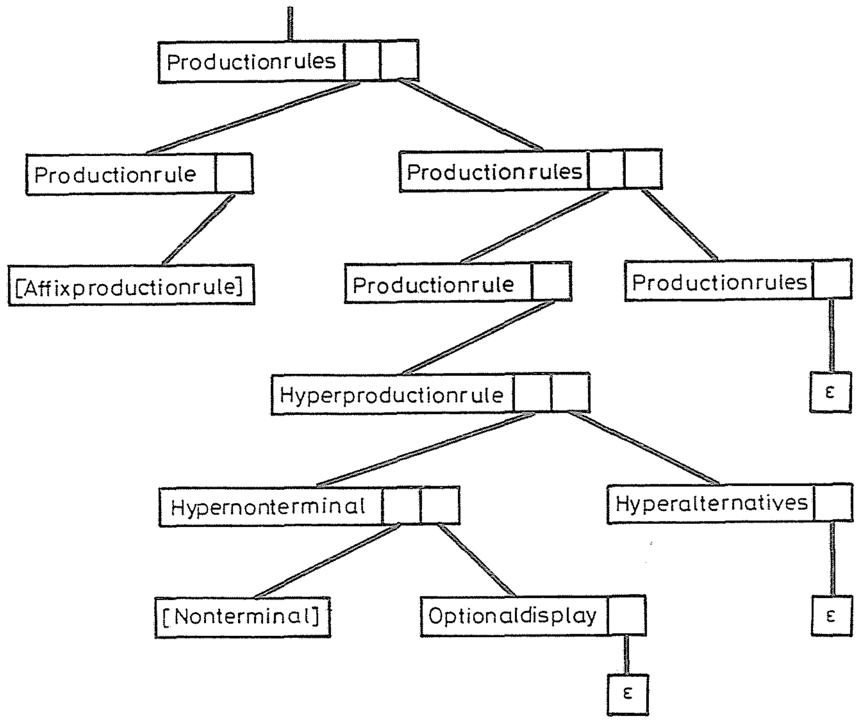

Fig. 4. Parse tree of the last parse

Eventually these eight parse trees are combined in one 3-dimensional parse tree shown in *Fig. 5.*

5.1.4 Selection Methods

For a text with untyped placeholders in it several different parse trees may be found. An untyped placeholder in this string is probably replaced in each parse tree by a different typed placeholder. The user of the editor can only work with the untyped placeholder if the unparsing (the textual representation) of the untyped placeholder is linked to all typed replacements.

The unparsing of an untyped placeholder which replaces only one typed placeholder is the unparsing of this typed placeholder. If it replaces several different typed ones the unparsing will be the symbol '□'.

The system offers the user a focus mechanism for selecting syntactical parts of the program. Focussing on some part of the program corresponds to selecting the corresponding subtree in the parse tree.

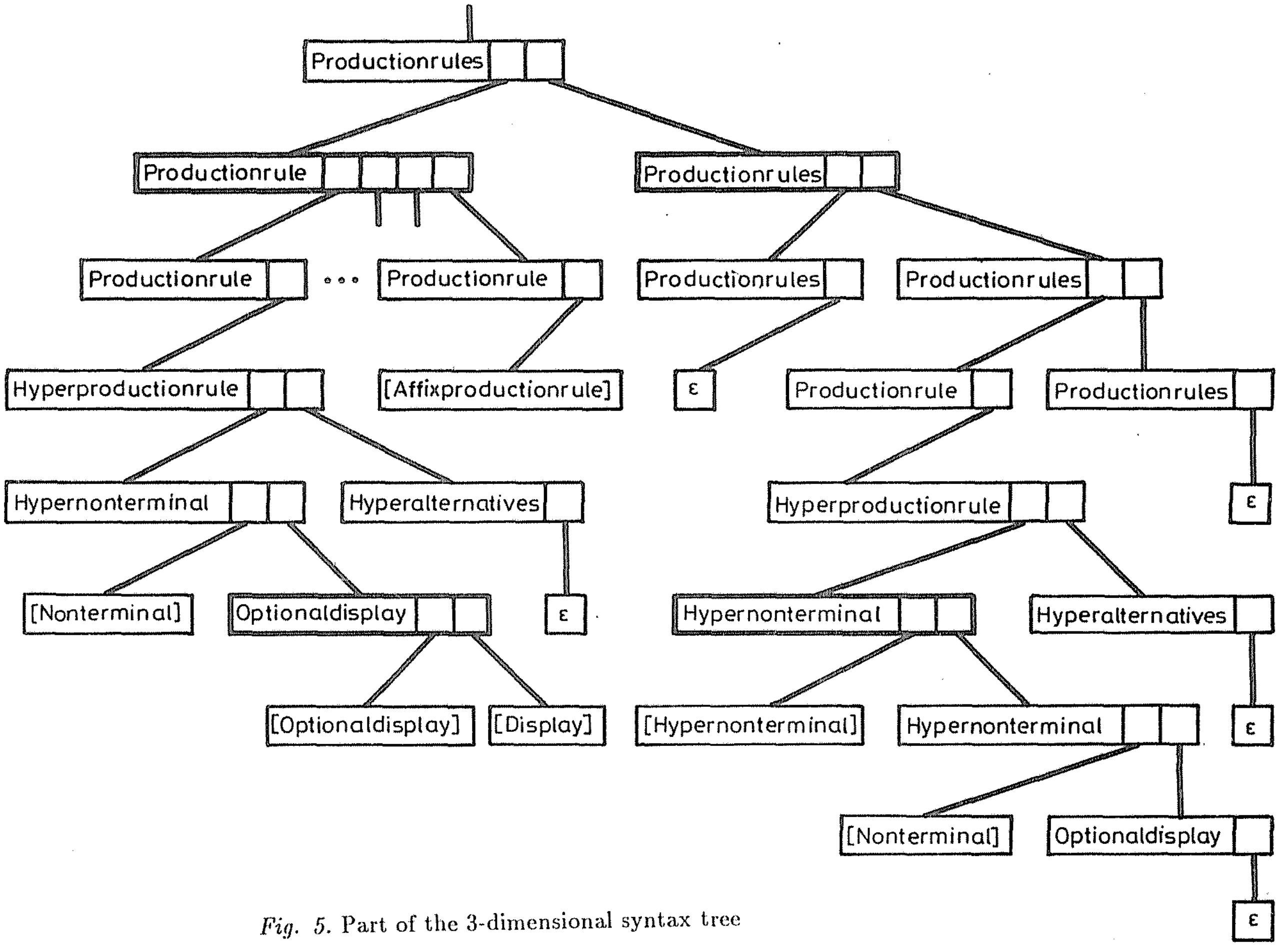

Fig. 5. Part of the 3-dimensional syntax tree

In order to make this focussing mechanism workable, each terminal symbol on the screen is connected to the corresponding node in the parse tree. The untyped placeholder symbols on the screen are connected to several nodes representing the typed placeholders in the subtrees in the parse tree.

Each untyped placeholder symbol on the screen is connected to a list which contains the links to the representations in the subtrees. A tree-traversal process stores the links in the lists.

If the user focuses on an untyped placeholder all possible typed placeholders are shown and the user can ask for all possible templates. A selection causes the replacement of an untyped placeholder by a typed one or a template. In both cases the subtrees, which are now superfluous, are removed from the parse tree.

5.2 Templates

In our system templates are not explicitly defined but derived from the context-free grammar of the specified language. The production rules are transformed into template specifications using the following rules:

1. the nonterminals on the right hand side are transformed into typed placeholders;
2. the terminals are not transformed;
3. commas and points are removed.

The right hand sides of all the alternatives of a production rule are templates for the nonterminal in the left hand side of the production rule. So, the nonterminal has a set of templates, denoted by $\mathcal{T}(A)$. The set of templates for the production rule:

```
hyper productionrule:
  hyper nonterminal,
    ":",
      hyperalternatives,
        ".".
```

is:

$\mathcal{T}$(hyperproductionrule)

= { [hypernonterminal] ":" [hyperalternatives] "." }

The occurrence of the nonterminal `layout` is considered as a special kind of nonterminal for which no typed placeholder is included in the derived template. The specification writer must define the layout and the places where the parser should skip layout by inserting this nonterminal on the right hand side of the alternatives.

To reduce the length of a specification a short hand notation is invented for production rules such as:

```
letters:
  letters,
    letter;
letters:
  letter.

letter:
  "a";
letter:
  "b";
letter:
  "c";
     :
letter:
  "z".
```

These so-called 'semi-terminals' are defined as:

```
letters:
  {abcdefghijklmnopqrstuvwxyz}+.
```

These semi-terminals are used for example in the definition of nonterminals such as `identifier` and `charsequence`. The right hand sides of production rules in which semi-terminals occur are not transformed into templates. The typed placeholders for the corresponding left hand sides can only be replaced by plain text.

It is possible to restrict the templates of a nonterminal to the templates obtained by the transformation of the right hand sides of the corresponding production rule. We think, however, that this is not optimal. In some cases a long derivation may be necessary to arrive at the construction the user has in mind. Consider the templates for the production rules `hypermembers`, `hypermember` and `hypernonterminal`.

```
𝒯(hypermembers) = { [hypermember];
                    [hypermembers] "," [hypermember] }
```

```
𝒯(hypermember) = { [hypernonterminal];
                   [terminals];
                   [hyperset] }

𝒯 (hypernonterminal) = { [nonterminal] [optionaldisplay] }
```

It will take the user three steps to transform the placeholder [hypermembers] into [nonterminal] [optionaldisplay].

This may be considerably reduced by deriving the templates of a production rule in a more efficient way. If the production rule has an alternative consisting of only one member, not being a semi-terminal, the set of templates of this nonterminal is also included in the set of templates of the production rule. This is done recursively, except for chain productions.

The sets of templates for the nonterminals hypermembers, hypermember and hypernonterminal using the strategy described above are thus:

```
𝒯(hypermembers) = { [nonterminal] [optionaldisplay];
                    "[charsequence]";
                    [set] [options] [display];
                    [hypermember];
                    [hypermembers] "," [hypermember] }

𝒯(hypermember) = { [nonterminal] [optionaldisplay];
                   "[charsequence]";
                   [set] [options] [display] }

𝒯 (hypernonterminal) = { [nonterminal] [optionaldisplay] }
```

6 Elegant Pretty Printing

Syntax-directed editor generator designers seem to find unparsing of limited interest, for the problem is usually solved by either:

- requiring unparsing rules in the specification, for example SSL, or
- doing no unparsing at all, for example in GIPE (HEERING and KLINT, 1986): the text is pretty printed in the same way as the user has inserted it.

We do not extend our specification formalism with extra unparsing primitives but want the generator to extract all unparsing information from the context-free grammar.

Our unparsing mechanism is based on the algorithm by (OPPEN, 1986). This algorithm works on a flat representation of the syntax tree extended with special symbols to direct the unparser. In articles of later date this possibility is further investigated and documented (ROSE and

WELSH, 1981; MATETI, 1983; RUBIN, 1983; LEAVENS, 1984; BAILES and SALVADORI, 1984; WOODMAN, 1986; JOKINEN, 1989).

The readability of a program text increased is according to Oppen, by spreading large syntactic program structures over several lines in a structured and consistent way. Small syntactic structures which fit on one line must not be split. Layout may only be inserted between the members on the right hand side of a production rule.

Oppen's algorithm has three passes. The first pass is a tree traversal mechanism to send the lexical and special symbols to a scanner. The second pass calculates the length of each syntactic structure. The last pass calculates the pretty print of the text.

The layout of some alternatives in the grammar has a strong 'horizontal' character whereas others have a 'vertical' character. If, however, a complete vertical structure fits in the remaining space on a line it will be printed horizontally. Members of alternatives which start and/or end with a terminal symbol are mostly printed on new lines at the same indentation as the first member of the alternative, so they have a 'horizontal' character. Members of alternatives which are left or right recursive are mostly printed after each other, so they have a 'vertical' character.

Our unparsing mechanism only traverses the syntax tree. First the length of each syntactic structure is calculated and stored in the corresponding node together with type information about the structure of the node. In the second tree traversal the pretty print is calculated using the type and length information.

An advantage of storing the length of each node is that after an alteration of the text only in a small part of the tree the length needs to be recalculated.

7 Conclusions and Further Research

The introduction of both types of placeholders increases flexibility. The resulting ambiguity turns out to be very local and can be tackled by the techniques presented in Subsection 4.2. We had thought to generalize this ambiguity and thus also generate editors for ambiguous grammars. Unfortunately, however, this conflicted with the incremental basis of syntax-directed editing.

The basis of our specification is EAG, in which it is possible to specify the semantics of the language. The nonterminals have sets of attributes to specify the (flow of) semantic information. Also a set of predicates is available for calculations. The evaluation mechanism used in our generated editors is described in (VAN DEN BRAND, 1990).

For the implicit extension of each production rule with two alternatives for recognizing placeholders the values of the attributes of the nonterminal in the left hand side of these new alternatives must be derived. These values are necessary to do as much attribute evaluation as possible. This is one of the problems we are currently investigating.

A prototype of our system is implemented. This implementation has to be improved and extended at several points. The user-interface is one of them; it must be made more flexible. If the user-interface is completed we will be able to compare our system with other systems.

Acknowledgements

We would like to thank Eerke Boiten, Kees Koster, Bert Windau, Erik Meijer, and especially Hans Meijer who made useful comments on the contents of this paper. Furthermore, we would like to thank Hugh Osborne for correcting our English.

References

AHO, A. V. - ULLMAN, J. D. (1972): The Theory of Parsing, Translation, and Compiling, Volume I: Parsing. Prentice-Hall, Englewood Cliffs, New Jersey.

BERGSTRA, J. A. - HEERING, J. - KLINT, P. (1989): Algebraic Specification. ACM Press in co-operation with Addison-Wesley.

VAN DEN BRAND, M. G. J. (1990): Incremental Affix Evaluation in Syntax Directed Editors. *CSN'90. Proc. Annual Conference on Computer Science in the Netherlands.* Utrecht, The Netherlands, 1–2 November 1990, pp. 35–51.

BAILES, P. A. - SALVADORI, A. (1984): A Semantically-based Formatting Discipline for Pascal. *Software—Practice and Experience*, Vol. 14, No. 3, pp. 235–251.

BAHLKE, R. - SNELTING, G. (1986): The PSG System: From Formal Language Definitions to Interactive Programming Environments. *ACM Transactions on Programming Languages and Systems,* Vol. 8, No. 4, pp. 547–576.

CAMPBELL, R. H. - KIRSLIS, P. A. (1984): The SAGA Project. *SIGPLAN Notices,* Vol. 19, No. 5, pp. 73–80.

HEERING, J. (1983): Taaldefinities Als Kern Voor Een Programmeeromgeving. In J. Heering and P. Klint, editors, *Colloquium Programmeeromgevingen,* Volume MC Syllabus 30, pp. 69–81, CWI, Amsterdam.

HEERING, J. - KLINT, P. (1986): A Syntax Definition Formalism. *Technical Report CS-R8633,* CWI, Amsterdam.

HEERING, J. - SIDI, J. - VERHOOG, A. (1986): Generation of Interactive Programming Environments - GIPE. *Technical Report CS-R8620,* CWI, Amsterdam.

JOHNSON, S. C. (1975): YACC - Yet Another Compiler-Compiler. *Technical Report Computer Science* No. 32, Bell Laboratories, Murray Hill, New Jersey.

JOKINEN, M. O. (1989): A Language-independent Prettyprinter. *Software—Practice and Experience,* Vol. 19, No. 9, pp. 839–856.

KLINT, P. (1983): A Survey of three Language-independent Programming Environments. *Technical Report CS-R83240,* CWI, Amsterdam.

KNUTH, D. E. (1968): Semantics of Context-Free Languages. *Mathematical Systems Theory,* Vol. 2, pp. 127–145, February 1968.

KNUTH, D. E. (1971): Semantics of Context-Free Languages: Correction. *Mathematical Systems Theory,* Vol. 5, pp. 95–96, May 1971.

KOSTER, C. H. A. (1975): A Technique for Parsing Ambiguous Grammars. In D. Siefkes, editor, *GI — 4.Jahrestagung, of Lecture Notes in Computer Science,* Vol. 26. pp. 233–246. Springer Verlag, Berlin/Heidelberg/New York.

LEAVENS, G. T. (1984): Prettyprinting Styles for Various Languages. *SIGPLAN Notices,* Vol. 19, No. 2, pp. 75–79.

LESK, M. E. (1975): Lex - A Lexical Analyzer Generator. *Technical Report Computer Science* No. 39, Bell Laboratories, Murray Hill, New Jersey.

MATETI, P. (1983): A Specification Schema for Indenting Programs. *Software—Practice and Experience,* Vol. 13, pp. 163–179.

MEDINA-MORA, R. (1982): Syntax Directed Editing: Towards Integrated Programming Environments. *PhD Thesis,* Carnegie-Mellon University.

MEDINA-MORA, R. - FEILER, P. (1981): An Incremental Programming Environment. *IEEE Transactions on Software Engineering,* Vol. 7, No. 5, pp. 472–482.

MEIJER, H. (1986): Programmar: A Translator Generator. *PhD Thesis,* Katholieke Universiteit Nijmegen.

MEIJER, H. (1990): The Project on EXTENDED AFFIX GRAMMARS at Nijmegen. In: Deransart, P. - Jourdan, M., editors, Attribute Grammars and their Applications *Lecture Notes in Computer Sciencs,* Vol. 26, pp. 130–143. Springer Verlag, Berlin/Heidelberg/New York.1990.

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

REPS, T. W. (1984): Generating Language-Based Environments. MIT Press.

REPS, T. W. - TEITELBAUM, T. (1989a): The Synthesizer Generator: a System for Constructing Language-based Editors. Springer Verlag, Berlin/Heidelberg/New York.

REPS, T. W. - TEITELBAUM, T. (1989b): The Synthesizer Generator Reference Manual. Springer Verlag, Berlin/Heidelberg/New York, Third edition.

RUBIN, L. F. (1983): Syntax-directed Pretty Printing—a First Step Towards a Syntax Directed Editor. *IEEE Transactions on Software Engineering,* Vol. SE-9, pp. 111–127.

ROSE, G. A. - WELSH, J. (1981): Formatted Programming Languages. *Software—Practice and Experience,* Vol. 11, pp. 651–669.

WATT, D. A. (1974): Analysis-Oriented Two-Level Grammars. *PhD Thesis,* University of Glasgow.

VAN WIJNGAARDEN, A. – MAILLOUX, B. J. – PECK, J. E. L. – KOSTER, C. H. A. – SINTZOFF, M. – LINDSEY, C. H. – MEERTENS, L. G. L. T. – FISKER, R. G. (1976): Revised Report on the Algorithmic Language Algol 68. Springer Verlag, Berlin/Heidelberg/New York.

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

Appendix A: A Simple Specification of EAG for Generation of an Editor

```
extendedaffixgrammar.

extendedaffixgrammar:
  layout,
    start nonterminal,
      productionrules (nil, defs).

productionrules (olddefs, defs):
  productionrule (olddefs, newdefs, defs),
    productionrules (newdefs, defs);
productionrules (defs, defs):   .

productionrule (olddefs, newdefs, defs):
  hyperproductionrule (olddefs, newdefs, defs):
productionrule (olddefs, olddefs, defs):
  affixproductionrule.

hyperproductionrule (olddefs, newdefs, defs):
  hypernonterminal (id),
    enter definition (id, olddefs, newdefs),
      ":", layout,
        hyperalternatives (defs),
          ".", layout.

hyperalternatives (defs):  hyperalternative (defs),
    ";", layout,
      hyperalternatives (defs);
hyperalternatives (defs):
  hyperalternative (defs).

hyperalternative (defs):
  hypermembers (defs);
hyperalternative (defs):
  .

hypermembers (defs):
  hypermember (defs),
    ",", layout,
      hypermembers (defs);
hypermembers (defs):
  hypermember (defs).

hypermember (defs):
  hyper nonterminal (id),
    check definition (id, defs);
```

```
hypermember (defs):
  terminals;
hypermember (defs):
  hyper set.

start nonterminal:
  nonterminal (id),
    ".", layout.

hyper nonterminal (id):
  nonterminal (id),
    optional display (nrofexps).

optional display (nrofexps):
  display (nrofexps);
optional display (empty):

display (nrofexps):
  "(", layout,
    affixexpressions (nrofexps),
      ")", layout.

affixproductionrule:
  affixnonterminal,
    "::", layout,
      affixalternatives,
        ".", layout.

affixalternatives:
  ";", layout,
    affixalternatives;
affixalternatives:
  affixexpression,
    ";", layout,
      affixalternatives;
affixalternatives:
  affixexpression.

affixexpressions (nrofafs + "i"):
  typed affixexpression,
    ",", layout,
      affixexpressions (nrofafs);
affixexpressions ("i"):
  typed affixexpression.

typed affixexpression:
  affixexpression;
typed affixexpression:
  ">", layout,
    affixexpression;
typed affixexpression:
  affixexpression,
    ">", layout.

affixexpression:
  affixterms.

affixterms:
  affixterm,
    "+" layout,
      affixterms;
affixterms:
  affixterm.

affixterm:
  affixnonterminal;
affixterm:
  affixterminals.

nonterminal:
  identifier (id),
    layout.

affixnonterminal:
  identifier (id),
    layout.

terminals:
  quotedstring,
    layout.

hyperset:
  set,
    options,
      display ("i").

set:
  "{",
    charsequence,
      "}", layout.

options:
  "+", layout;
options:
  "*", layout;
options:
  "+!", layout;
options:
  "*!", layout;
options:
  .

affixterminals:
  quotedstring,
    layout.

quotedstring:
  """",
    charsequence,
      """", layout.

enter definition (>id, >defs, newdefs):
 excludes (id, defs ),
    add to (id, defs , newdefs);
 enter definition (>id, >defs, defs):
  includes (id, defs).

check definition (>id, >defs):
  includes (id, defs).

excludes (>id, >nil):;
excludes (>id, >head + "+" + tail):
  not equal (id, head),
    excludes (id, tail).

includes (>id, >id + "+" + tail):;
includes (>id, >head + "+" + tail):;
  not equal (id, head),
    includes (id, tail).

add to (>id, >defs, id + "+" + defs):.

identifier (l + lgs):
  {abcdefghijklmnopqrstuvwxyz} (l):
    letgits (lgs),
      layout.

letgits (blanks + lgs1 + lgs2):
  { }*!  (blanks),
    {abcdefghijklmnopqrstuvwxyz1234567890}+!  (lgs1),
      letgits (lgs2);
letgits (empty):
  .

charsequence:
  ;
charsequence:
  {abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
  0123456789}*!  (chars),
    charsequence;
charsequence:
  {!@#$%^&*()_-+=|\\~‘[];:’<,>.?/}*!(char),
    charsequence.

layout:
  { \n}*!  (ign).
```

Address:

M. G. J. van den Brand
Department of Informatics
Faculty of Mathematics & Informatics
University of Nijmegen
Toernooiveld 1
NL-6525 ED Nijmegen
The Netherlands