PREGMATIC

A Generator for Incremental Programming Environments

een wetenschappelijke proeve op het gebied van de Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor aan de Katholieke Universiteit Nijmegen, volgens besluit van het College van Decanen in het openbaar te verdedigen op maandag 2 november 1992, des namiddags te 3.30 uur precies door

Marinus Gerardus Josephus van den Brand

geboren op 27 maart 1962 te Born

druk: Krips Repro Meppel

Promotores: Prof. C.H.A. Koster
Prof. dr. P. Klint, Universiteit van Amsterdam
Co-promoter: Dr. ir. H. Meijer

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Brand, Marinus Gerardus Josephus van den

Pregmatic : a generator for incremental programming
enviroments / Marinus Gerardus Josephus van den Brand. -
[S.l. : s.n.]
Thesis Nijmegen. - With index, ref.
ISBN 90-9005315-8
Subject headings: generators (computer science) /
incremental programming environments.

voor mijn moeder

Preface

The research on Pregmatic was financed by NWO, the Dutch Organization for Scientific Research; project number: 612-317-020.

I was confronted with Extended Affix Grammars (EAGs) [Mei86] for the first time during an advanced course on compiler construction. For this course Jan Damen, Hans Langeveld and I had to prepare a presentation which we chose to do on the generation of programming environments as in the thesis of Reps [Rep84]. Bert Windau and I (the 'gensde-twins') did our 'afstudeeropdracht'[1] on the combination of Extended Affix Grammars and programming environment generation, under the supervision of Hans Meijer. This established the basis for my future research on the subject.

In the two years that I was an 'onderzoeker in opleiding'[2] I developed some techniques for the automatic generation of error detection[3] [BLM89] in the compiler generator Programmar [Mei86]. Some of these result are included in this thesis. Subsequently I started working on the design and implementation of a generator for programming environments. I wanted to develop a complete system based on Extended Affix Grammars, without extending this formalism with all kinds of features intended to specify properties of the tools comprising the resulting environment. Furthermore, I was not prepared to restrict the class of languages for which an environment could be generated. In order to be able to process arbitrary languages I chose a very general parsing technique, viz. (left-corner) backtrack parsing. The efficiency of this type of parser is very poor. The parsers generated by Programmar [Mei86] use the affix-directed parsing principle, which is also incorporated in the parsers in the environments generated by Pregmatic. The consequence is that the affix value propagation mechanism is also based on backtracking, which results in severe computational overhead. However generality was preferred over efficiency during this research. Given a prototype of the resulting system efficiency can be tackled in a structured way. In particular the amount of backtracking can be restricted in both the parser and the affix value propagation.

The ambiguity of a context-free grammar may give rise to several syntax trees and affix graphs for some input sentence. These trees are combined into one more complex tree which enables the user of the generated environment to work with this tree without being forced to disambiguate during the parsing process or to select one of the yielded syntax trees.

[1] masters's thesis

[2] postgraduate research assistent

[3] This research was also financed by NWO; project number: 125-30-04.

Research on the generation of programming environments is only justified if it includes incrementality. Although a lot of research is being done in this field, it is still worthwhile to look at incrementality, certainly with respect to EAGs. The key element of Programmar [Mei86] was affix-directed parsing — I wanted to make this technique incremental.

Given this framework I developed a prototype system called PREGMATIC[4]. It served as a basic framework in which adaptations and extensions were easily included. One of the extensions of PREGMATIC was a tool[5] to influence the unparsing of the syntactical constructs of the context-free grammar.

The specification used to test the prototype was that of a toy language PICO. Its specification can be found in Appendix C. Several other languages were used as a case-study during the development of PREGMATIC. Only SASL is included in this thesis, see Chapter 7 and Appendix D.

Chapter 1 gives an overview of existing systems and formalisms. Some details of PREGMATIC will already be introduced at some points in this chapter. The rest of the thesis gives a full description of various features of the system.

In Chapter 2 EAG as a specification formalism is described. Various properties of EAGs are presented, as well as the notion of affix-directed parsing. The differences between the EAG-formalism as used for generating compilers and transducers, and the EAG-formalism as used for generating programming environments will be pointed out.

Chapter 3 describes the implementation of EAGs. Because a rather unusual type of parser is used a lengthy description of this parser and its derivation from the specification formalism is given. The process of building the syntax tree and the corresponding affix graph is also described. The concept of the affix graph is formalized and a number of routines are defined which are necessary for the description of the evaluation method. Incrementality is still not taken into consideration in the description of this method.

In Chapter 4 the structure of the generated environments is described. The user-interface is explained together with the derivation of some language dependent characteristics, such as templates and placeholders. This chapter concludes with a description of the method in which the unparsing rules are derived and of the tool provided for modifying the unparsing of a single language construct.

In Chapter 5 the incremental features of a generated programming environment are discussed. The recognition of placeholders, syntactic incremental reparsing and incremental extension of the type checking mechanism incorporated in the environments are discussed. This chapter concludes with a number of measures for increasing the efficiency of type checkers.

Chapter 6 is rather independent of the rest of the text. It discusses execution tools which could be included in the environments generated by PREGMATIC.

In Chapter 7 an EAG specification describing the language SASL is presented. It describes a number of interesting language features such as the famous offside rule (i.e., block structure is defined by indentation), and complex expressions.

[4]The name PREGMATIC was suggested by Franc Grootjen, the first four letters stand for PRogramming Environment Generator.

[5]The design and implementation of this tool was done by Paul Jones.

Dankwoord (Acknowledgements)

I must apoligize for not writing these acknowledgements in English. José persuaded me to write them in Dutch, so ...

Als eerste wil ik iedereen bedanken die op een of andere manier meegewerkt heeft aan de tot standkoming van dit proefschrift, maar niet expliciet hieronder genoemd wordt.

Ik wil de volgende mensen met name bedanken: Kees Koster en Paul Klint voor het begeleiden van dit proefschrift als promotoren. Hans Meijer als dagelijks begeleider en als vriend. Hans was altijd bereid vage gedachten aan te horen en deze via discussies te stroomlijnen en concretizeren.

De beide leden van de manuscriptcommissie Henk Alblas en Jan van Katwijk voor het lezen van dit proefschrift.

Arend, Daniel, Eerke, Eric, Franc, Janos, Joop, Marc, Maria, Mark-Jan, Matthias, Niek en Wim voor de discussies en de gedachtenwisselingen over mijn onderzoek.

Hans van Halteren en Nelleke Oostdijk voor de interesse in mijn onderzoek.

Arie, Casper, Dinesh, Emma, Frank, Huub, Jasper, Jan Heering, Jan Rekers, Paul Hendriks, Pum, Susan en Wilco (kortom de medewerkers van het GIPE-project in Amsterdam) voor het meedenken en de belangstelling in mijn onderzoek. De GIPE-werkbesprekingen waren een grote bron van inspiratie.

Huub Borst Pauwels en John van Krieken voor het ontwikkelen van het user-interface van de gegenereerde omgevingen. Paul Jones voor het testen van de gegenereerde omgevingen en het verbeteren van de unparsing faciliteiten. Hugh voor het corrigeren van mijn geschreven Engels. Greta en Ineke voor de secretariële ondersteuning.

Hans Langeveld als enige afstudeerder. Bert Windau als andere helft van de 'gensdetwins' (het begin van dit alles).

Erik Meijer als vriend en collega, de etentjes met Lilian en José hadden altijd een rustgevende werking op mij gehad. Eigenlijk was het een soort van uitlaatklep: even op alles en iedereen schelden.

Jolande en Gabriëlle voor hun bijstand in Goede Tijden en Slechte Tijden.

Jannet Wiggers voor het ontwerp van de omslag van dit proefschrift.

De familie, en in het bijzonder de beide Moeders.

José (als vriendin en echtgenote) zonder jou steun, bemoediging en vertrouwen was dit proefschrift nooit tot een goed einde gekomen.

Contents

Chapter 1

Introduction

The goal of the research discussed in this thesis was the development of a complete system for the generation of incremental interactive programming environments from Extended Affix Grammars (EAGs) [Wat74] called PREGMATIC. The present project is an extension of previous research in the context of Programmar [Mei86], a translator generator based on Extended Affix Grammars. We have investigated the possibility of using a *given* EAG, describing both the syntax and 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 the language definition.

Existing formalisms for the specification of programming environments are complex and they are strongly influenced by the problems of environment generation. With extant programming environment generators, usually based on attribute grammars or similar formalisms, it is still a major effort to construct an environment. All kinds of details have to be specified explicitly because most generators suffer from an ad hoc approach.

This research is characterized by the fact that no properties which are unique to the subparts of the programming environment need be specified.

Programming environments are not only interesting for programmers, but also for language designers. For example, a programming environment generator is a useful tool 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. The description should therefore be devoid of specifications which are exclusively needed for the generation of a programming environment.

The goal of this research is the generation of programming environments, given a simple specification formalism. The resulting environment of course must be as efficient as possible. A major part of the research was therefore into the incremental behaviour of the generated environments and on the development of efficient parsing and propagation algorithms, given the constraints posed above. Nevertheless, efficiency was not the most important aspect of our research — generality was more important. We also wish to stress that most results can also be applied to formalisms similar to EAGs, such as attribute grammars.

A system for generating programming environments consists of three main parts.

1. The formalism to specify the language for which the environment is generated.

2. The generator itself, which is actually the least interesting part.

3. The generated programming environment.

Our system PREGMATIC also consists of these three parts. Each of them will be briefly discussed in the next three sections. We conclude this chapter with a description of incrementality.

1.1 Specification formalisms

Various formalisms have been proposed for the description of the programming languages for which programming environments must be generated. These specification formalisms must be expressive, since they have to be read and written by human beings, and they must be suitable for an automatic implementation. In this section we will restrict ourselves to the description of the tools for the specification of properties of syntax and type checking. Tools for the specification of the dynamic semantics of the language are described in Chapter 6.

Most specification formalisms for generating programming environments are based on attribute grammars [Knu68], usually enriched with unparsing rules, tools for the definition of templates and a mechanism for the specification of the abstract syntax. For this reason, most of the resulting formalisms consist of several parts, such as a collection of syntax rules specifying lexical, concrete, and abstract syntax, a collection of semantic rules specifying static and dynamic semantics, and a collection of unparsing rules [Hee83, Kli83]. Older formalisms tend to consist of several unrelated parts whereas more recent formalisms try to offer a more unified approach. There is no single uniform specification formalism from which all parts of a programming environment can be generated.

In attribute grammars syntax and static semantics are defined by different means. The static semantics are usually defined by *semantic functions* using a programming language like C or Pascal. HAG [VSK89] is an attribute grammar based formalism in which this strong distinction is abolished.

The specification formalism SSL of the Synthesizer Generator [RT89a] is strongly based on attribute grammars; we will give a short description of this formalism in Section 1.1.1. The specification formalism for the PSG-system [BS85, BS86] is also based on attribute grammars; however, instead of semantic functions the formalism uses *context relations*. This formalism will be discussed in Section 1.1.3. The specification formalism of the ALOE-system [Med82] is based on *action routines* derived from the semantic routines used in compiler generators such as YACC [Joh75]. We will discuss the specification formalism of this system in Section 1.1.2.

A second way of specifying an environment is by means of an algebraic specification [BHK89]. The specification formalism ASF+SDF of GIPE [HKKL86, HSV86, Kli91] is based on this principle. The ASF+SDF formalism will be discussed in Section 1.1.4.

This is not an exhaustive list of specification formalisms but it represents the most important ways of specifying languages for the generation of programming environments.

There are a lot of very interesting formalisms which are not (yet) used for the specification of programming environments, such as HAG [VSK89].

Designers of programming 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 inferred from the specification of the input. Specification formalisms tend to become 'polluted' in several ways:

- adaptations, for example left-factorization of the underlying context-free grammar;

- extensions, for example extending the specification with templates.

If these two can be avoided by using powerful generators the result will be more elegant specification formalisms. We will not give an exhaustive list of unnecessary adaptations and extensions but will give an example of each in Section 1.1.5.

1.1.1 SSL

The Synthesizer Specification Language (SSL) was developed for the Synthesizer Generator [RT89a]. This system can be considered as the YACC [Joh75] of the programming environment generators. We will not give a detailed description of SSL but only a few characteristics.

The context-free *concrete syntax* of the specified language must be LALR(1), because the parser generator is based on YACC [Joh75]. The core of the formalism is an attribute grammar — almost each detail of the environment is specified by means of attributes. SSL consists of several parts:

- Rules for specifying the *abstract syntax*.

- Rules for the specification of attributes and attribute equations, which also include the rules for defining the error attributes and the specification of the semantic functions.

- *Unparsing rules.*

- The specification of the rules for translating the concrete syntax into abstract syntax.

- Specification rules for templates and transformations.

The framework of an SSL specification is the abstract syntax. The facilities for explicitly defining this abstract syntax make the generated environment quite flexible. The Synthesizer Generator [RT89a] is therefore also well-suited for generating transformation systems, such as the PROSPECTRA-system [KBHG$^+$87] and the BMF-system [VBF90].

1.1.2 The ALOE specification formalism

The generator in the ALOE-system [Med82] generates programming environments including template editors (see Section 1.3.1).

It is not necessary to specify the concrete syntax of a language for this type of editors. ALOE specifications are therefore smaller than corresponding specifications in other formalisms.

An ALOE specification consists of abstract syntax rules, priority rules, and unparsing rules. Besides these parts, which are more or less indispensable for the specification of the syntax of a language, the specification writer must also indicate for each node in the abstract syntax tree whether this node may be used as a 'file node', i.e. whether the subtree of this node may be stored as a file. Furthermore, the language name plus version number must be specified. These are necessary to guarantee that a file created with an editor is not processed by an editor generated for a different language or a different version of the same language. A file created by an ALOE generated editor contains not a textual representation of the program but a tree representation.

The semantics, both static and dynamic, are defined by means of action routines. These action routines are specified in a special-purpose programming language, GC, a dialect of C. Each production rule is extended with a number of action routines describing the derivation of the static semantics. These routines are also used to manipulate the tree, cursor, focus, and to report errors. The specification writer may create new language specific action routines by writing them in GC using the ALOE Implementation Environment.

1.1.3 PSG specification formalism

The PSG specification formalism is a formal non-procedural definition language consisting of four parts, viz. the definition of:

- the lexical, concrete, and abstract syntax of the language,
- the context conditions,
- the denotational semantics, and
- the unparsing rules.

There is a strict distinction between the different types of syntax. The lexical symbols in the specified language must be explicitly defined, but the lexical symbols for identifiers and integers, for example, are predefined. The concrete syntax rules are context-free grammar rules extended with transformation rules for the construction of the abstract syntax tree. Because the parser in a generated editor is an LL(1)-parser the concrete syntax must be LL(1). A node in the abstract syntax tree is defined by a 'class identification' and a 'construction rule'. These nodes may have either a fixed number of subparts of different syntactic types, or a flexible number of subparts of the same type.

Type checking in the PSG-system has the following characteristics:

- Arbitrary incomplete programs are type checked.
- Type errors are detected as soon as possible.
- Type checking is fully incremental.

The evaluation mechanism used in the system can be characterized as an attribute evaluation method method using unification and ignoring flow information. The specification writer need not have any knowledge of the underlying evaluation technique, he only has to define the context relations. These relations consist of three parts:

- The 'scope- and visibility rules' which define which occurrences of an identifier in a program fragment are related to each other. There is a predefined notion of scope rules.
- The 'data attribute grammar' which defines the abstract syntax of the attribute values used in the specified language.
- The 'basic relations' which define the attribute assignments related with each node in the abstract syntax tree. They define the local context conditions.

The PSG specification formalism has a strong modular structure which enables the specification writer to develop the syntactical properties of the language independently from the context conditions.

1.1.4 ASF+SDF

The ASF+SDF specification formalism is a combination of two independently developed formalisms:

- ASF, Algebraic Specification Formalism, and
- SDF, Syntax Definition Formalism.

The ASF part of the formalism is used to define the type checking and dynamic semantics of a language. ASF is a many-sorted algebraic specification formalism [BHK89, Hen91]. The implementation of the underlying algebraic specification is done by term rewriting. The formalism allows modular structuring of the specification. Specifications in ASF consist of several subparts, including export, import, and parameters sections.

The SDF part [HHKR89] serves to specify the context-free grammar of the language. Specification in SDF also contains several components, the three most important of which are:

- lexical syntax,
- context-free syntax, and
- priority rules.

The context-free syntax defines the concrete syntax. The abstract syntax cannot be specified explicitly — the system will always derive the optimal abstract syntax tree — but the specification writer can influence this construction by postfixing the production rules with extra information relevant to the concrete structure of the abstract syntax tree. The abstract syntax tree can be only influenced slightly. The SDF part offers no other tools for the specification of the abstract syntax. The constructed abstract syntax tree can be changed by means of functions which the specification writer has to define in the ASF part.

There are no possibilities for defining the unparsing of the abstract syntax tree; unparsers are generated automatically. During editing, the text is presented to the user in exactly the same form in which it was originally entered.

A third striking difference is that the underlying context-free grammar is not restricted to the classes LL(1) or LALR(1), but may be an arbitrary (even ambiguous) context-free grammar.

The formalism is thus quite different from the formalisms presented earlier.

1.1.5 Pollution

Most of the specification formalisms presented consist mainly of unrelated subparts. As well as requiring the explicit specification of various features of the programming environment the formalisms also suffer from unnecessary limitations. Our intention was to use a given specification formalism and derive a complete programming environment from it. The resulting environment will be less flexible than, for example, an environment generated by the Synthesizer Generator [RT89a], but the specification will be simpler and more uniform.

Adaptations

In some systems, the specification writer may have to rewrite the structure of the context-free grammar before transforming it into a specification. Possible rewritings are left-recursion elimination and left-factorization, if the parser is an LL(1)-parser. It is even possible that the specification writer may not be able to transform an arbitrary grammar into a specification because the parser in the syntax-directed editor of the generated environment is not powerful enough. Generators based on YACC [Joh75] only accept specifications for which the underlying grammar is LALR(1). It is not possible to generate an environment for languages which do not satisfy this condition. It may thus be impossible to experiment with syntax-directed editing based on ambiguous context-free grammars, or with the parsing of languages requiring arbitrary lookahead, however interesting this may be.

It may also be necessary to separate lexical and syntactical properties of the grammar. This separation is necessary in formalisms of systems based on LEX [Les75] and YACC [Joh75], because each of these tools has its own specification formalism.

By using a different type of parser and a more powerful generator these adaptations become superfluous in PREGMATIC.

Extensions

The specification writer may have to add text not directly related to the grammar in order to specify some part or tool in the environment, for example: templates, names of placeholders or unparsing rules.

Most syntax-directed editors allow the use of templates. In the SSL specification formalism [RT89b] these templates are defined explicitly — the templates for the non-terminal `<|exp|>` are, for example, defined as:

```
transform exp on "+" <|exp|>:  Sum(<|exp|>,<|exp|>),
              on "-" <|exp|>:  Diff(<|exp|>,<|exp|>),
              on "*" <|exp|>:  Prod(<|exp|>,<|exp|>),
              on "/" <|exp|>:  Quot(<|exp|>,<|exp|>)
              ;
```

This kind of information is redundant, since it is also implicitly available in the context-free grammar.

Again using a more powerful generator makes the specification of this information unnecessary in PREGMATIC. Even the abstract syntax need not be defined explicitly, however this may lead to a less flexible environment.

1.2 Programming Environment Generators

Programming environment generators can be considered as transducers from specifications to programming environments. The complexity of the transductions depends on the level of explicitness of the specification formalism. If each component of the environment is explicitly defined in the formalism a simple transducer suffices to generate the environment. Some of these systems nevertheless have interesting features with respect to programming environment generation. One system is GIPE [HK86] (Section 1.2.3). In Section 1.2.2 a few remarks will be made concerning the Synthesizer Generator [RT89a]. The remaining two systems mentioned, ALOE and PSG, are not discussed in this section.

1.2.1 PREGMATIC

The generators used in PREGMATIC are also transducers. There are two characteristics related to these transducers which may be worth mentioning. One of the prototypes of Programmar [Mei86] is used as transducer generator and as a result *all* generators in PREGMATIC are written as EAGs. The second one is that virtually all language specific properties of the environment are automatically derived from the specification. The generators automatically determine certain characteristics of the specification to be used during generation, see Chapter 4.

1.2.2 Synthesizer Generator

The Synthesizer Generator [Rep84, RT89a] is a generalization of the Cornell Program Synthesizer [TR81], a programming environment developed for a subset of PL/I.

The lexical scanner and the parser are generated using LEX [Les75] and YACC [Joh75] respectively. The remaining components of the environment are generated by simple hand-written transducers.

The SSL specification is extensively type checked and various properties of the specification are calculated. One of the tests performed on the underlying attribute grammar is the *orderedness test*. If the attribute grammar satisfies the ordered attribute grammar requirement [Kas91] a more efficient evaluation strategy can be used rather than the more general and less efficient algorithms described in [Rep84].

During the generation process the expressions in the specification are optimized by, among others:

- constant folding,
- short-circuited boolean expression evaluation, and
- tail-recursion elimination.

1.2.3 GIPE: ASF+SDF meta-environment

The GIPE (Generation of Interactive Programming Environments) project [HK86] is a long term research project funded by the European ESPRIT programme. It has resulted in a common toolkit called CENTAUR [BCD+89] that has been used to construct various specialized systems. One of them is the ASF+SDF meta-environment [Kli91] which we will discuss here.

Most systems transform the complete specification into an environment. This is a rather time consuming process which is annoying if a complete new environment must be generated every time a small alteration is made in a specification under development.

ASF+SDF meta-environment [Kli91] has a few interesting characteristics with respect to this generation process. The system is strongly based on lazy and incremental program generation techniques. This approach considerably speeds up the generation process but may slow down the parsing performance. Detailed descriptions can be found in [HKR87, HKR90, HKR91]. Instead of generating a complete scanner and parser, only those parts of the parser which are really needed during the parsing of an input sentence are generated (lazy parser and scanner generation). Modifications of the specification only affect the relevant parts of the parser. The previous parser is not thrown away but it is incrementally adapted. These techniques are applied to the generated LR(0)-parsers.

In earlier versions of the system the ASF part was translated into Prolog code [Hen91]. The ASF part is currently compiled into Lisp code [Wal91]; see Section 1.3.3.

1.3 Environments

We have described the specification formalisms and the programming environment generators without indicating exactly what must be specified and what must be generated. The notion of programming environments is extensive. An unrelated collection of language independent tools (for example an editor, a debugger, and a compiler) available on an arbitrary computer forms a programming environment. We are interested in environments consisting of *integrated* tools, where all tools use the same internal representation of the program and all tools are immediately available to the user of the environment. We will also restrict ourselves to environments which are *incremental*, this notion will be explained in Section 1.5. Furthermore, the environment must be *interactive*; batch-oriented development of programs is uneconomical. The programming environment will be restricted to one language, for a different language a new environment must be generated.

The typical components of an *incremental interactive programming environment* may be:

- an editor,
- a parser,
- a type checker,
- an unparser ('pretty printer'),
- an interpreter,
- a debugger,
- a compiler, and
- a library tool.

The nucleus of the programming environment from the user's point of view is the syntax-directed editor, but with respect to the internal operations it is the abstract syntax tree. All other tools are available through the editor. They must either be invoked explicitly or called implicitly. In the latter case the execution of the tool cannot be influenced by the user. Parser, type checker, and unparser do their job without intervention by the user. Both types of tools operate on the same abstract syntax tree.

The various tools will be discussed in the rest of this section, and for each of them we will indicate on which systems they are available. The organization of this section is directed towards the tools in the generated environments, rather than towards the systems.

1.3.1 Editor

A syntax-directed editor is the key element of each environment. The adjective 'syntax-directed' means that the editor has knowledge of the language for which it has been generated. This knowledge is used to prevent the introduction of errors, not only syntax errors but also type errors. The properties of the language are checked by means of a parser and a type checker, see Sections 1.3.2 and 1.3.3.

We distinguish two different classes of syntax-directed editors [Log88]: *template editors* and *text editors*. *Hybrid editors* combine features of both. Template editing is strongly based on the use of *placeholders* and *templates*. A placeholder is a special symbol representing a non-terminal which is to be defined later. A template is a framework, usually the right hand side of a production rule in the context-free grammar, in which the non-terminals are again replaced by placeholders.

Template editors The editor generated by the ALOE generator of the GANDALF system [MF81, Med82] is an example of a template editor. It is impossible to create syntactically incorrect programs with this kind of editor. The greatest drawback of template editors is that editing small-scale constructs like expressions is tedious. It is also usually not easy to replace one construct by another, for example a while-loop by an until-loop, unless such a transformation is explicitly defined in the specification.

Text editors The class of text editors we are considering does not include editors such as vi or emacs. Instead, text editors process the text almost immediately and transform it into a tree representation. This type of editor allows the modification of program text at arbitrary places. The user must only indicate which subpart of the text he wants to modify. The editor used in the environment of the SAGA system [CK84] is a text editor. The editor in the ELAN-programming environment [KW86] is also 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 [RT89a], the PSG-system [BS86], and the ASF+SDF meta-environment [Kli91] are hybrid editors. A hybrid editor offers the possibility of using both template editing and text editing. The disadvantages of the two types of editors can be circumvented, their advantages can be combined. The level of integration between both editing modes varies for each system.

The editors generated by PREGMATIC are also hybrid editors. They distinguish two different editing modes:

- The *template edit mode*, in which the user works with placeholders and templates.
- The *text edit mode*, in which the user is allowed to modify one syntactical construct[1] in his program.

[1] A syntactical construct is a piece of program which is derived from some non-terminal.

1.3.2 Parser

The parser in the programming environment is used whenever the user has modified a piece of program in text edit mode. It will be implicitly invoked to parse the input sentence, and either builds an abstract syntax tree and corresponding (attribute) graph if the parsing was successful, or reports an error if the parsing failed. Editors which do not support the text edit mode, for example the environments generated by the ALOE generator [Med82], need no parser.

Generated environments usually use LALR(1)-parsers, but LL(1)-parsers are also used, for example in the environments generated by the PSG-system [BS86]. Most programming languages can be described by these two types of grammars.

The use of a stronger type of parser may serve to increase the possibilities of the generated environment or to remove restrictions on the specifications. The ASF+SDF meta-environment [Kli91], for example, uses a generalized LR-parser [Rek92]. It can parse arbitrary (even ambiguous) context-free grammars.

The PREGMATIC-system uses a left-corner backtrack parser (Section 3.1), which also makes it possible to work with ambiguous context-free grammars. Furthermore, it enables us to introduce a new kind of placeholders (Sections 1.4 and 5.4) and to experiment with *affix-directed parsing* in connection with incremental techniques. The notion of affix-directed parsing will be explained in Section 2.3.

1.3.3 Type checker

A type checker is included in all three types of syntax-directed editors. After an abstract syntax tree and the corresponding graph are constructed the program being edited is type checked by the type checker.

Most of the research in the area of programming environments is concerned with these type checkers, and concentrates on the development of efficient incremental type checkers.

Synthesizer Generator

In [Rep84] optimal-time change-propagation algorithms were presented, which are incremental extensions of attribute evaluation methods. The type checkers generated by the Synthesizer Generator [RT89a] are based on these algorithms. This system uses various incremental attribute evaluation methods. If the attribute grammar in the SSL is ordered, an incremental ordered attribute grammar evaluator is used, which is more efficient than the algorithms presented in [Rep84].

Various incremental attribute evaluators have been developed (an overview can be found in [Alb91a]). Initially, an attribute graph is constructed and evaluated. As a result of an edit action a subgraph is replaced by a new subgraph and thus attribute values may have changed. Consequently, the attributes of a number of graph nodes must be recalculated. The main idea of these incremental attribute evaluators is that the number of graph nodes to be visited by the evaluator in order to restore consistency of the attribute graph is minimized.

PSG

The type checkers generated by the PSG-system [BS86] are based on attribute grammars, context relations, and unification. Because of the use of unification they are more powerful than the type checkers of environments based on incremental attribute evaluation.

Rather than working with a single attribute value, the type checker works with sets of still-possible attribute values. This makes it possible to type check arbitrary incomplete programs. As soon as some set of attribute values becomes empty the type checker stops because a type error has been detected.

The incremental behaviour of the type checker is obtained by attaching local relations to each tree node, which are only relevant for the subtrees of this node.

ASF+SDF meta-environment

The type checker in the environments generated by the ASF+SDF meta-environment [Kli91] is based on a completely different evaluation strategy, it uses term rewriting instead of attribute evaluation. The equations in the ASF part are translated to rewrite rules, which operate from left to right. A Prolog implementation of this mechanism is described in [Hen91]. A Lisp implementation is described in [Wal91].

Initially the type checker did not support incremental evaluation, because algebraic specifications are in general less suited to it. In [Meu90] a subclass of the algebraic specifications is defined, viz. the 'conditional well-presented primitive recursive schemes' which *are* suited to incremental evaluation. A well-presented primitive recursive scheme is isomorphic to a strongly non-circular attribute grammar. The nodes in the abstract syntax tree are extended with attributes in which the values of reduced terms are stored. An incremental attribute evaluation algorithm is used to determine which terms must be evaluated after a modification, and therefore which attributes must be re-evaluated. This algorithm is a modified version of the algorithm presented in [RTD83].

1.3.4 Unparser

The unparser is responsible for transforming the abstract syntax tree into a readable textual representation. In most specification formalisms the unparsing is specified by means of special unparsing rules for each type of node in the tree.

The unparser is a tree traversal algorithm which unparses each node in the tree according to the rules specified. The unparsing may also depend on the space remaining on a line. The layout of the output produced cannot be influenced by the user of the environment unless he alters the unparsing rules and generates a complete new environment.

In the ASF+SDF meta-environment [Kli91] the unparser cannot be influenced at all by the user or the specification writer because the unparsing rules are automatically derived from the syntax definitions.

The environments generated by PREGMATIC include a tool for modifying the unparsing of syntactical constructs, see Section 4.3.4.

1.3.5 Interpreter and debugger

Most systems for the generation of programming environments have a tool for the execution of the developed programs. A few even support the interactive debugging of programs. Although the implementation in each system may be quite different, three major techniques for executing programs are used [Kai89].

1. *Action routines*, which are applied in the programming environments generated by GANDALF [HN86]. The execution of a syntactical construct can be specified by means of an action routine, which will be stored in the corresponding tree node The execution of a program corresponds to the execution of the action routines in each tree node of the corresponding abstract syntax tree

2. *Attribute grammars* are mostly used to describe the static semantics of a language using semantic equations. An intermediate code which will be executed by a language independent interpreter is necessary in order to describe the dynamic semantics. The Synthesizer Generator [RT89a] is based on this technique.

   In [WJ88] attribute grammars are extended with so-called gate attributes to make them suitable for the specification of dynamic semantics using the normal attribute evaluation techniques as much as possible. Description of dynamic semantics in ordinary attribute grammars is impossible because of the cycles which may be introduced. The gate attributes in the modified attribute grammars of [WJ88] identify these cycles. These gate attributes also ensure the use of two different evaluation techniques, viz. one for evaluating the attributes outside the cycles and one for evaluating attributes within a cycle.

3. *Denotational semantics* are used in the environments generated by PSG [BS86].

This list can be extended with a fourth strategy: the environments generated by ASF+SDF meta-environment [Kli91] use *term rewriting* for the execution of the developed programs. The dynamic semantics of the language is specified in the equation part of the ASF+SDF formalism [Hen91]. Algebraic functions in the equation part are evaluated using term rewriting (Section 1.1.4). In Figure 1.1 we will give a small part of the ASF+SDF specification of the dynamic semantics of the `whilestatement`.

```
module eval
  ⋮
  exports
  ⋮
    "evs"     "(" SERIES "," VALUE-ENV ")"       -> VALUE-ENV
    "evstat"  "(" STATEMENT "," VALUE-ENV ")"    -> VALUE-ENV
    "eve"     "(" EXP "," VALUE-ENV ")"          -> VALUE
  ⋮
  equations
```

```
  ⋮
[Ev5]       eve (Exp, Value-env) = false
            ===================================================
            evstat (while Exp do Series od, Value-env) = Value-env

[Ev6]       eve (Exp, Value-env) = true
            =========================================================
            evstat (while Exp do Series od, Value-env) =
              evstat (while Exp do Series od, evs (Series, Value-env))
  ⋮
```

Figure 1.1: `whilestatement` specification in ASF+SDF.

In [Kai89] a combination of action routines and semantic equations yielding action equations is described. These action equations are used for both the static and dynamic semantics of a language.

All strategies discussed so far, except for the ASF+SDF-approach, use techniques based on:

- a list of instructions that has to be evaluated [Kai89],
- abstract code which is executed by an interpreter [BS86, RT89a, NS91], or
- an attribute graph which is visited by an evaluator and yields the evaluation of the program [WJ88].

If the code is abstract code, as in the second case above, two different strategies can be used. Either a language independent evaluator executes the abstract code, or the interpreter is specified within the formalism.

In [NS91] the abstract code is the concrete code of the program. This concrete code is transformed into a predicate, which makes it possible to execute recursive syntactical constructs, such as the body of a loop, quite elegantly. Although they claim to be working without intermediate code, they are using the concrete code for this purpose.

The key element of execution is again incrementality. Both interpreter and debugger must support the execution of incomplete programs and arbitrary program fragments. The execution of an incomplete program is stopped as soon as a placeholder is encountered and the editor is automatically invoked to enable the user to extend his program.

A debugger is a more complex tool than an interpreter due to the interaction with the user of the environment. The user must be able to inspect the values of variables and to indicate which program fragments he wants to trace in more detail. A detailed description of the debugger generated by the PSG-system [BS86] can be found in [BMS87]. The debugger generated by the ASF+SDF Meta-environment [Kli91] is described in [Tip91].

In Chapter 6 we will discuss various possibilities to execute programs in PREGMATIC.

1.3.6 Libraries

A library within a generated environment may be very handy in order to have simple facilities to store and retrieve program fragments, other than ordinary file-IO-facilities. Only the PSG-system [BS86] offers a language-independent library facility.

1.4 Placeholders

We have discussed several types of syntax-directed editors in Section 1.3.1. The hybrid editor proved to be the most flexible one. We believe however that it can be made even more flexible.

Editors generated by the Synthesizer Generator [RT89a] make it possible to edit the text of a single syntactical construct using text edit mode. This selected construct may contain several placeholders. The parser is called immediately after leaving the text edit mode. The changed program text is rejected if it still contains placeholders, because the parser cannot recognize these placeholders. Thus text containing placeholders created by the template facility of the editor cannot be recognized by the parser of the same editor.

This inflexibility is caused by too sharp a distinction between text edit mode and template edit mode. Manipulating placeholders in text edit mode, other than replacing them by plain program text, is forbidden. It would be more elegant to allow the user to introduce placeholders in text edit mode.

A placeholder, consisting of special open and close brackets enclosing the name of the replaced non-terminal, will be called a *typed placeholder*. We also introduce the *untyped placeholder*; a new kind of terminal symbol not associated with any specific non-terminal.

The users of the editors generated by PREGMATIC are allowed to manipulate both typed and untyped placeholders in text edit mode. They can both be modified and inserted. This extension makes the text edit mode more flexible.

1.5 Incrementality

Research on generating programming environments is motivated by the exploration of the incremental behaviour of such environments. Most implementations of incremental systems are based on 're-using as much as possible', without consideration of its costs or whether storing all intermediate results is useful. However, there is a trade-off between re-use and recalculation.

We can distinguish three cases:

- the program is syntactically and static semantically correct,
- the program contains a syntax error, and
- the program is syntactically correct but contains some type error.

With respect to incrementality the last case is particularly interesting. Systems such as the Synthesizer Generator [RT89a] will detect the type error and report it, but they will also build the corresponding abstract syntax tree and attribute graph because, after correcting the error, parts of the syntax tree might be re-used. This is a rather ad hoc solution. It is possible to build a unique abstract syntax tree and attribute graph because the underlying context-free grammar is not ambiguous. There is no guarantee that some part of this syntax tree can be re-used after correcting the error.

In EAGs and, more specifically, in systems based on affix-directed parsing a program is correct if it is correct with respect to both syntax and type checking. Only if both constraints are satisfied will the corresponding syntax tree and affix graph be constructed, the latter of which will also be consistently decorated.

This does not imply that the user of the environment is forced to correct his erroneous programs. The erroneous program fragments will be included in the syntax tree, not as a subtree but as text. The realization of these ideas can be found in the Sections 4.2.1 and 5.2.3. Correction of the error implies complete reparsing of the erroneous program fragment.

The underlying context-free grammar may be ambiguous, but the affix-directed parsing eventually ensures that only one solution will be found. If the program text contains some type error this may prevent the affix-directed parsing mechanism resolving the ambiguity and thus several syntax trees could be built. In order to make the system workable one of these must be selected, which is as arbitrary as selecting the text oriented approach.

Chapter 2

Extended Affix Grammars

Extended Affix Grammars [Wat74] are a member of the family of two-level grammars. They are a direct offspring of two-level van Wijngaarden grammars [WMP+76]. EAGs are less general than van Wijngaarden grammars but can be implemented more efficiently. The version of EAGs used for PREGMATIC is an extended subset of the EAGs used for Programmar [Mei86].

EAGs are an extended form of affix grammars [Kos71], which were invented in 1962 by Koster and Meertens [MK62] to describe a subset of the English language. The extension from which they derive their name is the possibility of using affix expressions at parameter positions.

In attribute grammars [Alb91b], attributes are either inherited or synthesized. In EAGs affixes are inherited and derived respectively. The inherited affixes are denoted by "`>affix`" and the derived ones by "`affix>`". The inherited affixes in the left hand side and the derived affixes in the right hand side of a production rule are called 'defining', the other ones are called 'applying' affix occurrences.

Attribute grammars and EAGs are in fact strongly related. There are two main differences between the two formalisms. The first one is purely syntactical and has to do with the notation of the affixes. The attributes in attribute grammars are explicitly named and their values are transferred by means of assignations. The second difference is that in attribute grammars the operations on the attribute values are performed by functions outside the formalism. In EAGs the domain of the affix values is strings. The only operations on these values are the equality test, concatenation and its inverse: splitting a string in several parts. It is possible to split an affix value by writing an affix expression at a defining affix position. One of the consequences of this symmetry of operations is that it is possible to leave out the flow symbols, viz. the "`>`"-symbol, in the specification of production rules.

A tutorial on EAG can be found in [Mei90] and an example contrasting EAG with other two-level formalisms in [Kos91b].

2.1 $\mathtt{a}^{\mathtt{n}}\mathtt{b}^{\mathtt{n}}\mathtt{c}^{\mathtt{n}}$ as an EAG specification

As an example we take the language $L = \{\mathtt{a}^{\mathtt{n}}\mathtt{b}^{\mathtt{n}}\mathtt{c}^{\mathtt{n}} | \mathtt{n} \geq \mathtt{0}\}$. The following eag can be developed for this language.

For the sake of clarity we will use the term EAG for the formalism and the term eag when we refer to a particular specification in the EAG-formalism.

```
anbncn:
  as (n),
    bs (n),
      cs (n).

as (zero):  .
as (one + na):
  "a",
    as (na).

bs (zero):  .
bs (one + nb):
  "b",
    bs (nb).

cs (zero):  .
cs (one + nc):
  "c",
    cs (nc).
```

```
zero ::  "".

one ::  "b".

n ::  zero;
      one + n.

na ::  n.

nb ::  n.

nc ::  n.
```

While this eag does not show all characteristics of the formalism it is useful for the explanation of some of them. An eag can be split into two levels, see Figure 2.1. On the first level we only have the underlying context-free grammar. Grammar, rules, non-terminals, and sets on the first level are extended with the notion 'hyper', if they are connected to the displays on the second level. Every notion describing something on the second level is prefixed by the notion 'affix'.

The eag consists of four *hyper rules* (the rules in the left column) and six *affix rules* (the rules in the right column).

The first hyper rule in the left column specifies the structure of the input sentence. It should consist of a string of a's, a string of b's, and a string of c's, the number of a's, b's, and c's should be equal. The second, third, and fourth hyper rule specifies each of these strings. The affix rules in the right column specify the domains used on the second level. In the example the domain of the affix values are strings consisting of b's. The number of a's, b's, and c's recognized in the input are unary counted. The empty string represents zero, whereas the string "b" represents one. The other rules in the right column describe arbitrary unary values.

The hyper rules consist of either one *alternative*, for example `anbncn`, or more than one alternative, for example `as`. An alternative of a hyper rule consists of a *left hand side*, for example `anbncn` and a *right hand side* which consists of zero or more *members*. These members may be either *terminals* or *hyper non-terminals*. A terminal is, as usual, a string enclosed by double quotes, such as `"a"`. A hyper non-terminal is a non-terminal which may be followed by a *display*, such as `as (zero)`. A display is a sequence of one or more *affix expressions*. Each affix expression consists of *affix terms* and the operators `+` or `*`. An affix term consists of *affix terminal*, *affix non-terminal*, *affix set*. The operator `+` is either a concatenation operator or an addition operator. In the example eag the `+` is used to concatenate string values. The `*`-operator, the tuple operator, can be used to build data structures. Both operators will be explained in Section 2.2.5. For example, `one + na` is an affix expression, but it may also be written as `"b" + na`, or as `{b} + na` where `"b"` is an affix terminal and `{b}` is an affix set respectively. The left hand side of a hyper rule consists of a hyper non-terminal.

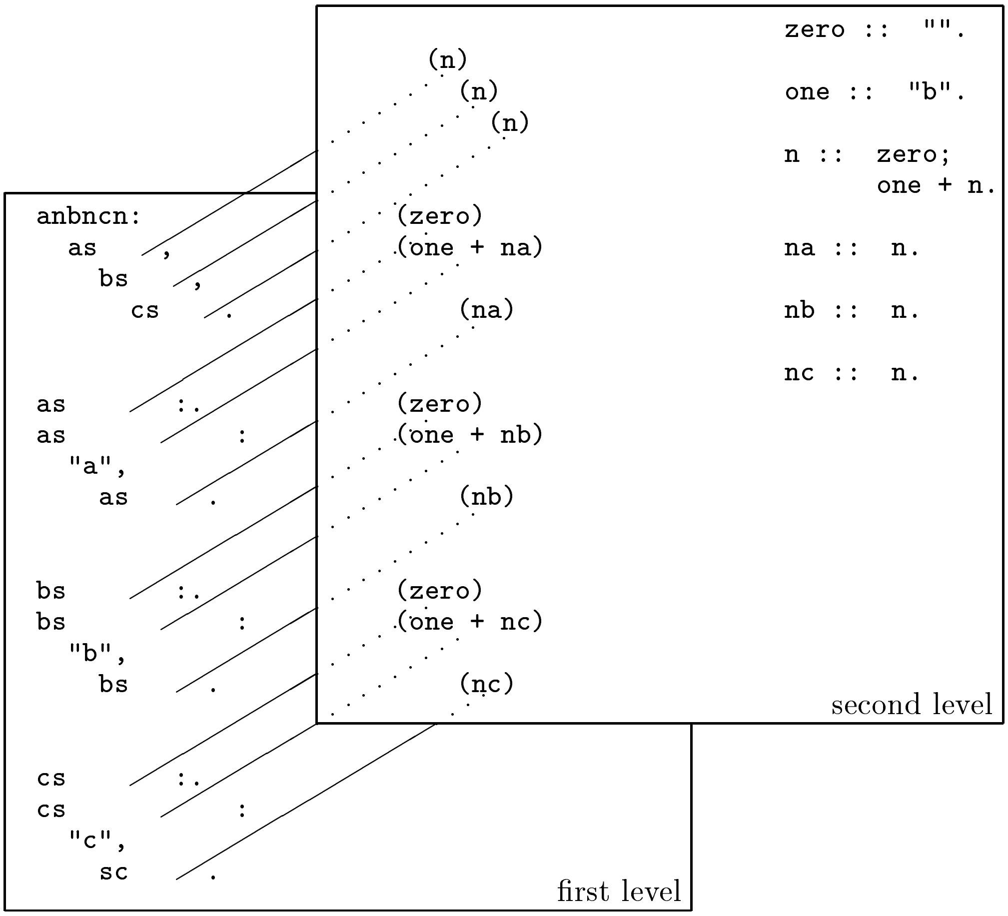


Figure 2.1: Levels of an eag.

As a notational extension, the hyper rule for `bs` may also be written as:

```
bs (nbs):
  {b}*!  (nbs).
```

The {b} in the right hand side of the hyper rule is a *hyper set*. The postfix *! of the set means that the longest possible sequence of b's must be recognized in the input (this will be explained in Section 2.2.2). The hyper sets must be followed by a display containing exactly one affix expression via which the recognized string will be returned.

An affix rule also consists of a left hand side, for example n, and one or more affix expressions, for example zero and one + n in n. The affix non-terminal n could also be defined as:

```
n ::  {b}*!.
```

An affix rule defines the domain of the affix non-terminal in left hand side.

This description covers only the syntax of an eag but there is also some interesting semantics associated with an eag. The *consistent substitution constraint*, which is derived from the notion of consistent substitution in two-level van Wijngaarden grammars [WMP+76], demands that each occurrence of an affix non-terminal in an alternative of a hyper rule represents the same value. The hyper non-terminals in the right hand side of the hyper rule anbncn are all extended with a display containing the affix expression n. The consistent substitution constraint demands that each occurrence of n represents the same value. This constraint could be formulated explicitly in the following way:

```
anbncn:
  as (ans),
    bs (bns),
      equal (ans,bns),
        cs (cns),
          equal (bns,cns).
```

where equal is a *predicate occurrence*. A predicate is a kind of semantic function (the notion of predicates will be further explained in Section 2.2.1). This predicate equal is a so-called *primitive* predicate. The hyper rule for cs may also be written as:

```
cs (n):
  {c}*!  (ncs),
    equally long (ncs, n).
```

The hyper non-terminal equally long in the hyper rule cs is also a predicate but it can be defined within the formalism itself as:

```
equally long (>zero, zero):
  .
equally long (>"c" + rcs, one + ncs):
  equally long (rcs, ncs).
```

The reason for the flow symbol (>) in the first affix expression of both displays will be explained in Section 2.2.1.

2.2 Formal definition of EAG

The formal definition of the EAGs as used in Programmar, Programmar-EAGs, can be found in [Mei86, Mor89]. It admits a number of extensions to the original EAG-formalism [Wat74]. We will give a formal definition of the variant of EAGs which we use for the generation of programming environments, PREGMATIC-EAGs, which differ from Programmar-EAGs with respect to affix flow and affix domain types. The eag presented in the previous section is a PREGMATIC-EAG.

In the rest of this thesis the notion EAG stands for PREGMATIC-EAG.

An eag consists of a affix grammar[1], an (underlying) context-free grammar, and a set containing information about the non-terminal occurrences and their associated affix expressions. First we will describe the affix grammar; it defines the domains of certain affix non-terminals:

$$G_{AG} = (N_A, T_A, S_A, P_A)$$

where
N_A is the finite set of *defined affix non-terminals.*
T_A is the finite set of *affix terminals.*
S_A is the finite set of *affix sets.*
P_A is the finite set of *affix rules*:

$$P_A \subset N_A \times (N_A \cup T_A \cup S_A)^*$$

As well as defined affix non-terminals N_A there are *free* affix non-terminals, which are *not* defined. The set of these non-terminals is F_A. V_A is the finite set of all affix non-terminals, $V_A = N_A \cup F_A$. The intersection of the sets of free- and defined affix non-terminals should be empty. The notion of affix sets will be explained in Section 2.2.2. Given this affix grammar and the free affix non-terminals we are able to define affix expressions:

$$AE \subset (V_A \cup T_A \cup S_A)^*$$

The (underlying) context-free grammar can be defined as:

$$G_{CFG} = (N, T, S_H, P, B)$$

where
N is the finite set of *non-terminals.*
T is the finite set of *terminals.*
S_H is the finite set of *hyper sets.*
Non-terminals can be extended with displays containing affix expressions, this extension yields the *hyper non-terminals.* H is the set of these hyper non-terminals.

$$H \subset N \times AE^*$$

For example, the hyper non-terminal `hyper (meta + free + {set} + "terminal")` is member of the set H. The notion of hyper sets will be explained in Section 2.2.2.

[1]Note that this affix grammar is different from the affix grammars introduced by Koster [Kos71].

B is the initial hyper non-terminal.
Given the sets of hyper non-terminals, terminals, and hyper sets we are able to define P, the set of *hyper rules*:

$$P \subset H \times (H \cup T \cup S_H)^*$$

Given the sets H, N and AE we are able to define the hyper non-terminals more precisely:

$$n(ae) \in H \Longleftrightarrow n \in N \wedge ae = ae_1, \ldots, ae_{\mathcal{N}_n} \text{ with } ae_i \in AE$$

$\mathcal{N}_M$ represents the number of affix expressions associated with non-terminal M. Given this number $\mathcal{N}_M$ and the non-terminal M we are able to define the set K describing non-terminal occurrences and their associated affix expressions as follows:

$$K = \{(x, \mathcal{N}_x) | x \in N\}$$

Given G_{AG}, G_{CFG}, and K we are able to define the full tuple representation of an eag.

$$G_{EAG} = (N_A, T_A, S_A, F_A, P_A, N, T, S_H, K, P, B)$$

In Appendix A we will give a complete eag for a representation of the EAG-formalism which incorporates a consistency check on the number of affix expressions associated with each hyper rule.

As an example we will write the eag for the language $\mathtt{a^n b^n c^n}$ given in Section 2.1 as a tuple. The hyper rule for `bs` will be replaced by the hyper rule with the hyper set in the right hand side.

```
G_EAG = (#N_A=#{zero,one,n,na,nb,nc},
         #T_A=#{"b"},
         #S_A=#{},
         #F_A=#{nbs},
         #P_A=#{zero ::  "".,
                one ::  "b".,
                n ::  zero; one + n.,
                na ::  n.,
                nb ::  n.},
         #N=#{anbncn,as,bs,cs},
         #T=#{"a","b","c"},
         #S_H=#{{b}*!},
         #K=#{(anbncn,0),(as,1),(bs,1),(cs,1)},
         #P=#{ anbncn:  as (n), bs (n), cs (n).,
                as (zero):  .,
                as (one + na):  "a", as (na).,
                bs (nbs):  {b} (nbs).,
                cs (zero):  .,
                cs (one + nc):  "c", cs (nc).},
         #B=#anbncn)
```

2.2.1 Predicates and consistent substitution

The type checking of a language being described can be specified explicitly by EAG predicates, or implicitly by the consistent substitution constraint. A *predicate* is (by definition) a hyper non-terminal which either generates empty or fails, depending on the values of its affixes. A predicate P can be either *primitive*, such as `not equal`, or defined in terms of other predicates like

$$
\begin{array}{l}
\mathtt{P}_0 : \mathtt{P}_{11}, \ldots, \mathtt{P}_{1p_1}. \\
\quad\vdots \\
\mathtt{P}_0 : \mathtt{P}_{n1}, \ldots, \mathtt{P}_{np_n}.
\end{array}
$$

where each of the $\mathtt{P}_{ij}$ is a predicate. Predicates play the same rôle as the semantic functions of attribute grammars. Affix-directed parsing is obtained by evaluating these predicates *during* syntactical analysis (Section 2.3).

There are no flow symbols ("`>`") in an eag unlike the directions in attribute grammars, except for the *critical affix positions* in predicates, i.e. those which are necessary for the evaluation of the predicate. If a predicate has several alternatives the same affix positions in the left hand side of the distinct alternatives must be marked as critical.

Suppose an eag includes the following rule:

```
identifierlist (deflist):
  identifier (name),
    identifier definition (name,restdeflist,deflist),
      ",",
        identifierlist (restdeflist).
identifierlist (deflist):
  identifier (name),
    identifier definition (name,nil,deflist).
```

where the predicate `identifier definition` is defined as follows:

```
identifier definition (>name,>deflist,newdeflist):
  excludes (name,deflist),
    add to (name,deflist,newdeflist).
```

The affix positions with the affix expressions `name` and `deflist` are the critical affix positions of this predicate. The affix position with the name `newdeflist` is not a critical one, because this affix contains a value which is the result of the execution of the predicate `add to`:

```
add to (>name,>deflist,name*deflist):
  .
```

The critical affix positions act as a kind of semaphore (which may cause delayed evaluation) and they should be specified in such a way that termination of the evaluation of predicates is ensured. Consider the hyper rule `identifierlist`: the predicate `identifier definition` is called before the rest of the identifier list is recognized. The result is that the predicate cannot be evaluated because the value of the affix non-terminal `restdeflist` is not yet known. The evaluation of this predicate is *delayed* until the rest of the identifier list is processed and the value of the affix non-terminal `restdeflist` is available. A *delayed predicate* is considered as a predicate whose evaluation has not, at this moment, failed so that the execution of the rest, either evaluation or parsing, can proceed. When the value of a critical position of the delayed predicate becomes available the evaluation of this predicate will be reconsidered.

A predicate remains delayed until *all* critical affix positions have a value. The specification of which affix positions are critical is quite important. Forgetting one could lead to non-termination. Suppose the predicate `excludes` in `addto` is defined as:

```
excludes (>id1,id2*env):
  not equal (id1,id2),
    excludes (id1,env).
excludes (>id,nil):
  .
```

If the value of `id1` is available before the value of the second affix position the evaluation of this predicate will never stop even though `not equal` is delayed.

Every function can be specified by means of predicates. We wish to demonstrate their power with respect to the specification of efficient type environments. Predicates can be used to build and manipulate complex data structures such as binary trees, in a flexible way.

```
empty tree ::  nil.

replace (>key,>info,>empty tree,
          empty tree*key*info*empty tree):
  .
replace (>key,>info,>left*key*tinfo*right,
          left*key*info*right):
  .
replace (>key,>info,>left*tkey*tinfo*right,
          newleft*tkey*tinfo*right):
  smaller (key,tkey),
    replace (key,info,left,newleft).
replace (>key,>info,>left*tkey*tinfo*right,
          left*tkey*tinfo*newright):
  smaller (tkey,key),
    replace (key,info,right,new right).
```

We assume that the predicate `smaller` in this predicate `replace` is primitive. We have considered the possibilities of allowing brackets in an affix expression, since an expression of the form `left*(key*info)*right` make the structure of the tree more explicit, but felt no real urge to introduce them.

The specification of type checking in EAG is primarily based on consistent substitution, a notion which was introduced in two-level van Wijngaarden grammars. Consistent substitution demands that affix occurrences with the same name within an alternative of a hyper rule represent the same value. Consider the hyper rule:

```
picoprogram:
  program start (programname),
    declarations (deflist),
      series (deflist),
        program end (programname).
```

The multiple occurrence of the affix non-terminal `programname` in `program start` and in `program end` imply that the rule `picoprogram` will only succeed when the value of both affix non-terminal occurrences is the same.

2.2.2 Semi-terminals

In Section 2.1 we introduced the notion of affix- and hyper sets.

```
bs (nbs):
  {b}*!  (nbs).
```

The hyper sets are in fact a shorthand notation. It is possible to define terminals, such as identifiers, numbers etc, with these sets. The hyper non-terminals in the left hand side of those hyper rules with a set in the right hand side are therefore called *semi-terminals*. The set of semi-terminals is defined as:

$$ST = \{N \mid N \to \alpha x \beta \in P \wedge x \in S_H\}$$

The hyper non-terminal `bs` is member of the set ST.

An affix set

```
letter::
  {abcdefghijklmnopqrstuvwxyz}.
```

is a shorthand notation for an affix rule of the form:

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

The exact syntax of both affix- and hyper sets can be found in Appendix A.

The display of the hyper set should contain precisely one affix expression. The parser returns the actual value read in the input by way of this expression.

The options which may follow both the hyper- and affix sets may need some extra explanation. We explain the options only for the hyper sets, they can also be defined in a similar way for the affix sets. The hyper set $\{a_1 \ldots a_n\}$ is defined by the following rule, in which we consider $\{a_1 \ldots a_n\}$ to be a non-terminal:

```
{a₁ ... aₙ}:
  "a₁".
  ⋮
{a₁ ... aₙ}:
  "aₙ".
```

The +-option can be defined as:

```
{a₁ ... aₙ}+:
  {a₁ ... aₙ}.
{a₁ ... aₙ}+:
  {a₁ ... aₙ},
    {a₁ ... aₙ}+.
```

Whereas the *-option can be represented as:

```
{a₁ ... aₙ}*:
  .
{a₁ ... aₙ}*:
  {a₁ ... aₙ},
    {a₁ ... aₙ}*.
```

The !-option specifies that the longest possible sequence of elements matching the elements in the set should be recognized. The !-option cannot be defined by some rule. The use of the !-symbol makes the set strict, viz. only the longest possible sequence is tried. Leaving out this option means that *all* prefixes of the longest possible sequence of elements will be tried as well.

The sets can be used, for example, for the specification of `layout`:

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

For the specification of special characters, such as `newline`, we use the C escape convention.

2.2.3 Defined affix non-terminals

The defined affix non-terminals are specified by the grammar $G_{AG} = (N_A, T_A, S_A, P_A)$. These defined affix non-terminals describe their affix value domains, which are used in two ways:

- as a mechanism to check whether the value assigned to a defined affix non-terminal is a member of the language described by parsing the value
- as a mechanism to generate values which are members of the described language.

In both Programmar-EAGs and PREGMATIC-EAGs, affix non-terminals may be used that are not defined, the so-called free affix non-terminals. In this case their domain is not restricted. Defined affix non-terminals are treated with great care in Programmar-EAGs. One of the well-formedness conditions formulated in [Mei86] states that defined affix non-terminals describing infinite languages should not be 'applying-only' which it is if it is at an applying affix position and it does not occur in a defining affix position within the same alternative.

Since flow is not explicitly specified we do not have the notion of defining and applying affix occurrences. The notion of applying-only defined affix non-terminals therefore does not exist but the defined affix non-terminals may nevertheless cause problems during the affix evaluation process in our system. The primary function of the affix grammar is the specification of the domains of affix values of the affix non-terminals occurring in the left hand side of the affix rules. These defined affix non-terminals check whether affix values assigned to them during evaluation belong to their language. However, in order to obtain a fully decorated affix graph it may be necessary for some of the defined affix non-terminals to generate elements of their language. We will call these non-terminals *generative defined affix non-terminals*. If a defined affix non-terminal describing an infinite language were to start generating values the affix evaluation process would never stop. These affix non-terminals *must* only have a recognizing function during evaluation, but defined affix non-terminals describing finite languages may enumerate all elements of their language during the affix evaluation process (this will be discussed in Section 3.4.6).

2.2.4 Well-formedness

In [Mei86] three well-formedness conditions were formulated for Programmar-EAGs. These conditions hold for PREGMATIC-EAGs as well. An eag is well-formed if the following three conditions are fulfilled.

1. The eag may contain no cycles unless they are blocked by affix values in some way.
2. Termination of the predicates must be guaranteed.
3. Generative defined affix non-terminals must have finite languages.

An eag has a cycle if non-terminal A properly rewrites to A. This leads to non-termination during parsing. If however hyper non-terminal A has an affix position with a finite domain which is continuously decreasing or increasing the termination is ensured by the affix evaluation process. Such an affix position is quite similar to the critical affix positions of the predicates and is therefore also called critical. It is denoted in the same way: by prefixing it with a flow symbol. The affix evaluation mechanism will treat these critical affix positions in almost the same way as the critical ones of the predicates (Section 5.2).

The specification writer must guarantee the termination of the predicates as described in Section 2.2.1 by indicating which affix positions are critical.

The notion generative is less obvious for PREGMATIC-EAGs than the notion of applying-only is for Programmar-EAGs. The specification writer should be very careful in using defined affix non-terminals. Termination with respect to defined affix non-terminals is guaranteed by the implementation. However either a lot of unnecessary work may be involved or the evaluation process may end prematurely.

2.2.5 Type checking EAGs

In Section 2.2.4 three well-formedness conditions were formulated. Violating one of these conditions leads to non-termination during evaluation. Whether an eag satisfies these conditions cannot be determined during the transformation from eag to programming environment. In this section we will describe a number of conditions which have to be fulfilled in order make an eag well-typed. It is our goal to check these conditions before an environment is generated.

The first condition has to do with identification. For each applied hyper non-terminal a corresponding definition should exist, unless the hyper non-terminal was primitive. For each affix non-terminal in the right hand side of an affix rule a definition should exist, unless this affix non-terminal was primitive. Thus, no free affix non-terminals are allowed in the right hand side of affix rules. Note that application before definition is allowed. The second condition states that all occurrences of a hyper non-terminal should have a consistent number of affix positions. These two conditions are straightforward and easy to check. The third condition is more complicated and will be described in the rest of this section.

The affix values in EAGs as described in [Mei86] and [Kos91b] are strings only. In PREGMATIC, the EAG-formalism is extended with numerals and tuples. We distinguish 3 types: *STRING*, *NUMERAL* (≥ 0), and *TUPLE*. The `*`-operator is associated with the *TUPLE* type.

The two new types, *NUMERAL* and *TUPLE*, are also available in the Programmar-EAGs, but there they are considered syntactic sugar. The effect of syntactic sugaring is that the concatenation operator is not only polymorphic but affix values of 'different' types are allowed on both sides of the operator. It is possible to write an affix expression like: `"abc" + 123`, or `"abc" + a * b`. It is not possible to define an illegal affix expression in Programmar-EAGs with respect to the concatenation operator and the types of its operands. The extensions of the affix value domain are not considered as syntactic sugar in PREGMATIC-EAGs. As a consequence we need a type checking mechanism.

In order to be able to reason about type checking we have to introduce some definitions to assign types to hyper- and affix rules. Furthermore, we have to indicate under which conditions a hyper- or affix rule is well-typed. For example, do we consider the following affix rule as a well-typed affix rule?

```
A ::  1 + A;
      "a" + A.
```

We will give three different type models, each of them has consequences for the type checker. Before we describe these three models, we have to introduce some definitions.

Definitions

The affix expression `"ab" + "c"` will generate the value `"abc"`, whereas the affix expression `"ab" + c` accepts values which consist of the prefix `"ab"` and some suffix. The tuple operator in an affix expression `a*b*c` generates an affix value which is a tuple consisting of 3 other affix values. This affix expression `a*b*c` *only* accepts affix values consisting of tuples of 3 elements. An affix expression only accepts the assigned value if it is of the appropriate type. An affix expression consisting of only one affix non-terminal accepts *every* value, unless this is a defined affix non-terminal, in which case the value must be member of the language defined by this affix non-terminal.

First, we introduce some notation.

Given the set of types the type of an 2-tuple is the same as the type of an 3-tuple. This mapping may be too general, and to be more specific about the types of the tuples we have to refine the basic type $TUPLE$ into the basic types $2-TUPLE$, $3-TUPLE$, ... , and $n-TUPLE$. There will always be a finite number of tuple types. We distinguish the following basic types for the affix values:

- $STRING$;
- $NUMERAL$;
- $2-TUPLE$, $3-TUPLE$, ... , $n-TUPLE$.

The basic types can be combined resulting in an *union* of (basic) types, this will be denoted as

$$(TYPE_1 \cup TYPE_2 \cup \ldots \cup TYPE_n)$$

The hyper- and affix rules can be considered as functions. An affix rule is an 0-ary function (viz. it has no arguments) with one of the basic types as result type.

```
zero ::  0.
```

is a function of type

$$zero : \Rightarrow NUMERAL.$$

The affix positions of a hyper non-terminal in the left hand side of a hyper rule are considered as the 'formal' arguments of the function represented by the hyper rule. The affix expressions on these position determine the type of the arguments. The result type of the function is always the basic type *STRING*, because such a function can be considered as a recognizing function on the input[2]. The hyper rules are $\mathcal{N}_N$-ary functions, with $\mathcal{N}_N$ the number of affix positions associated with the non-terminal N.

```
rule ("string", 100):  ...  .
```

is a function of type

$$rule :\ STRING \times NUMERAL \Rightarrow STRING.$$

The primitive predicates `equal` and `not equal` are polymorphic.

The affix non-terminals `empty` and `nil` are primitive. The affix non-terminal `empty` represents the empty string, whereas the affix non-terminal `nil` represents an empty n-tuple. Furthermore, the digits `0` through `9` are primitive, and 2 or more digits denote the usual value within the decimal system. The affix non-terminal `empty`, and the numbers $\{\mathtt{0}, \mathtt{1}, \ldots\}$ have the following types:

$$empty : \Rightarrow STRING.$$

$$0 : \Rightarrow NUMERAL.$$

$$1 : \Rightarrow NUMERAL.$$

$$\vdots$$

Because of the extended set of tuple types the affix non-terminal `nil` is polymorphic.

$$nil : \Rightarrow (2{-}TUPLE \cup 3{-}TUPLE \cup \ldots \cup n{-}TUPLE).$$

Concatenation of affix values of different types is *not* allowed. Furthermore, the concatenation operator is only defined within the affix domain types *STRING* and *NUMERAL*, concatenating tuples is not allowed. The following affix expressions are *not* allowed: `"abc" + 123`, or `"abc" + a * b`. Furthermore, it is obvious that the `+`-operator is overloaded:

$$\mathtt{+} : STRING \times STRING \Rightarrow STRING.$$

$$\mathtt{+} : NUMERAL \times NUMERAL \Rightarrow NUMERAL.$$

[2]The affix rules *never* have an immediate effect on the input.

To obtain the type of an arbitrary affix expression we define the function τ.

$$\tau(ae) = \begin{cases} \mathit{NUMERAL} & \text{if } ae \in \mathbb{N} \\ \mathit{STRING} & \text{if } ae \in S_A \vee (ae \in T_A \wedge ae \notin \mathbb{N}) \end{cases}$$

$$\tau(ae) = T \text{ if } ae \in N_A \wedge ae : \Rightarrow T$$

$$\tau(ae) = (\mathit{STRING} \cup \mathit{NUMERAL} \cup 2{-}\mathit{TUPLE} \cup \ldots) \text{ if } ae \in F_A$$

$$\tau(ae_1 \texttt{*} \ldots \texttt{*} ae_n) = n{-}\mathit{TUPLE}$$

$$\tau(ae_1 \texttt{+} \ldots \texttt{+} ae_n) = \begin{cases} \mathit{STRING} & \text{if } \forall i \leq n : \tau(ae_i) = \mathit{STRING} \\ \mathit{NUMERAL} & \text{if } \forall i \leq n : \tau(ae_i) = \mathit{NUMERAL} \end{cases}$$

For all $x \in S_H$ the type of the affix expression in the display is *always* STRING. Now, we are able to give the three type models of the EAG-formalism.

First type model

This model is the most restricted one. Using this model an eag is well-typed if:

- All alternatives of an affix rule have the same type.
- The i^{th} affix position in the left hand sides of all alternatives of a hyper rule is of the same type.

These two restrictions can be reformulated as 'union-types are *not* allowed'. Each affix- and hyper rule can be uniquely typed. One of the consequences is that tuples with a different number of elements cannot be mixed freely in the right hand side of an affix rule. The following affix rule is not well-typed.

```
env ::  nil;
        id * type * env;
        id * type * value * env.
```

In Section 2.2 the set K was introduced. K contained information about the number of affix positions associated with each hyper non-terminal. This can be considered as a very restricted form of type information. Now we are able to derive more information about the types of an eag and therefore we have to extend this set K.

$$K = K_H \cup K_A$$

where

$$K_H = \{(x, \mathcal{N}_x, \mathcal{T}_x) | x \in N\}$$

and

$$K_A = \{(x, \mathcal{T}_x) | x \in N_A\}$$

$\mathcal{N}_x$ in the set K_H is the number of affix positions.
$\mathcal{T}_x$ in the set K_H is a $\mathcal{N}_x$-tuple where each element t_i of the $\mathcal{N}_x$-tuple corresponds with the type of the affix expression on the i^{th} affix position.
$\mathcal{T}_x$ in the set K_A is one of the basic types.

The initial $\mathtt{a^nb^nc^n}$-eag has the following set K:

$$
\begin{aligned}
K = \{&(\mathtt{anbncn}, 0, ()),\\
&(\mathtt{as}, 1, (STRING)),\\
&(\mathtt{bs}, 1, (STRING)),\\
&(\mathtt{cs}, 1, (STRING)),\\
&(\mathtt{zero}, STRING),\\
&(\mathtt{one}, STRING),\\
&(\mathtt{n}, STRING),\\
&(\mathtt{na}, STRING),\\
&(\mathtt{nb}, STRING),\\
&(\mathtt{nc}, STRING)\}
\end{aligned}
$$

An element from the set K_H for this eag can also be written as:

$$x : \mathcal{T}_x \Rightarrow STRING$$

Similarly, an element from the set K_A can be written as:

$$x : \Rightarrow \mathcal{T}_x$$

An affix rule such as:

```
env ::  nil;
        id * type * env.
```

is well-typed and its type can be represented as:

$$env : \Rightarrow 3\!-TUPLE$$

or in the set K as $(\mathtt{env}, 3-TUPLE)$.

Each rule in the eag can be uniquely typed. Type errors caused by assigning an affix value of type *STRING* to an affix expression of type *NUMERAL* can be detected immediately.

The free affix non-terminals cause some extra overhead, because the type checker probably needs a closure computation to determine the type of these affix non-terminals. Free affix non-terminals can in some way be considered as polymorphic. It will be obvious that, given these 'severe' restrictions, it will be possible to define a static type checker for the EAG-formalism.

Second type model

This model is less restrictive than the previous one. An eag will be considered well-typed in this model if:

- All alternatives of an affix rule have the same type, except in that the alternatives need not be of the same tuple type.
- The i^{th} affix position in the left hand sides of all alternatives of a hyper rule must be of the same type, but again these affix positions need not be of the same tuple type.

These restrictions can be reformulated as 'union-types are *only* allowed with respect to the tuple types'. This allows affix rules such as:

```
env ::  nil;
        id * type * env;
        id * type * value * env.
```

We can use the set K such as defined for the first model, but the elements of the $\mathcal{N}_x$-tuple $\mathcal{T}_x$ in the set K_H and $\mathcal{T}_x$ in the set K_A may be a union of different tuple types.

The affix rule `env` is well-typed:

$$env :\Rightarrow (2\!-\!TUPLE \cup 3\!-\!TUPLE)$$

or as an element in K: $(\texttt{env}, (2\!-\!TUPLE \cup 3\!-\!TUPLE))$.

Allowing a union of tuple types does not mean that an 2-tuple affix expression accepts affix values generated by an 3-tuple affix expression, or vice versa. The number of tuple elements of an affix value can be used at runtime to select specific alternatives of a hyper rule. If the specification writer uses an 3-tuple at an affix position instead of an 2-tuple and the hyper rule has more than one alternative this error cannot be detected statically. This type error can only be detected at runtime. At compile time the type checker can only determine whether the set of tuple types for the i^{th} affix position of an applied hyper non-terminal is a subset of the set of tuple types defined by the i^{th} affix position of the same hyper non-terminal in the left hand side of a hyper rule.

It is still possible to define a static type checker for the EAG-formalism if it satisfies the restrictions formulated above, although it will be more restricted.

Third type model

This model does not restrict the types of

- the alternatives of the affix rules, or
- the i^{th} affix position in in the left hand sides of all alternatives of a hyper rule.

Type checking the eag at compile time will no longer be feasible, because every hyper- and affix rule will be well-typed. The type checker can only check whether the operands of the +-operator are of the appropriate type. Furthermore, it can check whether the set of types for the i^{th} affix position of an applied hyper non-terminal is a subset of the set of the types defined by the i^{th} affix position of the definition of this hyper non-terminal. Consider the applied non-terminal N:

$\ldots$, $N(AE_1, \ldots, AE_n)$, $\ldots$

The hyper rule defining this non-terminal will be:

$N(AE_1^1, \ldots, AE_n^1)$: $\ldots$.
$\vdots$
$N(AE_1^m, \ldots, AE_n^m)$: $\ldots$.

The type of non-terminal in the left hand side will be:

$$N: \ (\tau(AE_1^1) \cup \ldots \cup \tau(AE_1^m)) \times \ldots \times (\tau(AE_n^1) \cup \ldots \cup \tau(AE_n^m)) \ \Rightarrow STRING.$$

The application of the hyper non-terminal N will be well-typed if:

$$\forall i : \tau(AE_i) \subseteq (\tau(AE_i^1) \cup \ldots \cup \tau(AE_i^m))$$

The amount of type checking will be very limited.

If each rule would be extended with a type specification a static type check would become possible.

[N:$T_1 \times \ldots \times T_n$]
$N(EA_1^1, \ldots, EA_n^1)$: $\ldots$.
$\vdots$
$N(EA_1^m, \ldots, EA_n^m)$: $\ldots$.

The hyper rule N is well-typed if:

$$\forall i : (\tau(EA_i^1) \cup \ldots \cup \tau(EA_i^m)) \subseteq T_i$$

This extension of the formalism does not fit in with our goal of keeping the formalism as simple as possible.

Final remarks

In the beginning of this section the affix rule

```
A ::  1 + A;
      "a" + A.
```

was presented with the question whether this rule was well-typed. Given the three models we are now able to answer this question. In the first and second model this rule is *not* well-typed. The third model allows affix rules of this type, the type of the rule above will be:

$$A : \Rightarrow (STRING \cup NUMERAL).$$

We make the assumption that a type checker first determines the types of both alternatives before it assigns a type to the non-terminal `A`, otherwise this rule will not be well-typed.

To make sure the user of a generated environment is *never* confronted with a mysterious type checking message an approach described in the first or second model should be chosen.

The current implementation of the prototype is based on the third model, but if the system is extended with a static type checker this will be based on the second model.

2.3 Affix-directed parsing

The principle of affix-directed parsing will be explained independently of the parsing technique used (Section 3.1). A lot of research has been done in the area of attribute directed parsing [AMT91], but as yet with only a few results.

The principle itself is simple: during the recognition of the input sentence as much of the semantics as possible is considered. In an eag this is done by evaluating the predicates during the parsing of the input sentence. In this way type-incorrect programs can be rejected as soon as possible without doing a lot of unnecessary work such as parsing the rest of the program and building an abstract syntax tree. Of course this will not lead to a speedup in all cases but it is a convenient extension of the parsing process. A lot depends on the way in which the type checking is specified.

Affix-directed parsing can also be used to influence the parsing process. This is very useful for recognizing certain context-depending syntactical constructs, such as the offside rule in Miranda [Tur90]. Affix-directed parsing can also be used to disambiguate a context-free grammar.

```
prio ::  1; 2.

expr:
  term (1),
    reloper,
      term (1).
expr:
  term (1).

reloper:  "=".
reloper:  "<>".

term (prio):
  term (prio + 1),
    oper (prio),
      term (prio).
```

```
term (prio):
  term (prio + 1).
term (3):
  "-",
    term (1).
term (3):
  variable access.
term (3):
  "(", expr, ")".

oper (1):  "+".
oper (1):  "-".
oper (2):  "*".
oper (2):  "/".
```

This is demonstrated by simplified Pascal expressions (given above), which, if we leave out the affixes, form an ambiguous context-free grammar. Using affix-directed parsing ensures that each sentence is recognized in a non-ambiguous way.

The reason for explaining this principle here is that the combination of affix-directed parsing and incremental evaluation is new. However, in some cases it may seem that by using affix-directed parsing the incrementality is not fully exploited (Section 1.5).

In attribute grammars the semantic functions are specified by means of a separate language, for example C or Pascal. In the EAG-formalism there is no syntactical distinction between predicates and other hyper rules. Syntax and semantics are fully integrated, which makes it easy to write and read EAG specifications. Besides this integration on the syntactical level we also managed a full integration of the evaluation of the semantic functions and the parsing process.

2.4 Interpretation model of EAG

In [Mei86] the computation model of Programmar-EAGs is described using a *translator function* and *decorated parse trees.* The computation model of PREGMATIC-EAGs is slightly less complicated because of the absence of the translator function. The notation and terminology used in this section is strongly based on Section 2.2. In order to reason easily about occurrences of rules we introduce the same notation as used in [Mei86].

$X_{\setminus Y}$ denotes the set of all z such that $z \in X$ and $z \notin Y$.

If $X_1, \ldots, X_n$ are sets, where $n \geq 0$, the expression $X_1 \times \ldots \times X_n$ denotes the set of all ordered sequences $\langle x_1, \ldots, x_n \rangle$ of length n, where $x_i \in X_i$ for $1 \leq i \leq n$. Such a sequence will be denoted as $\langle x_i \mid 1 \leq i \leq n \rangle$, or as $\langle\rangle$ if $n = 0$. The abbreviation $\langle\!\langle x_i \mid 0 \leq i \leq n \rangle\!\rangle$ stands for $\langle x_0, \langle x_i \mid 1 \leq i \leq n \rangle\rangle$.

The set of affix terminals, affix non-terminals, and affix sets will be denoted by $A = (T_A \cup V_A \cup S_A)$. The set of terminals, hyper non-terminals, and hyper sets will be denoted by $U = (T \cup H \cup S_H)$.

The function $\mathcal{N}_x$ is defined as $\mathcal{N}_x : U \rightarrow \mathbb{N}$ with $\mathcal{N}_x(v) = 0$ if $v \in T$ and $\mathcal{N}_x(v) = 1$ if $v \in S_H$.

Let W be the set of all $\langle v, \langle ae_j \mid 1 \leq j \leq \mathcal{N}_x(v)\rangle\rangle$ in $V \times AE^*$. Let $A_T = (T_A \cup S_A)$ and $U_T = (T \cup S_H)$. Let $AE_T = A_T^*$ and R be the set of all elements in W which are also in $U \times AE_T^*$. AE represents an affix expression and AE_T an affix value. In an affix expression $\langle u_k \mid 1 \leq k \leq m \rangle$, u_k is its k^{th} affix term.

$\langle\!\langle x_i \mid 0 \leq i \leq m \rangle\!\rangle$ represents either an affix rule or a hyper rule, x_0 is the left hand side, $\langle x_i \mid 1 \leq i \leq m \rangle$ the right hand side and x_i , for $1 \leq i \leq m$ its i^{th} member.

The language of an element $a \in A$ is denoted by $L(a)$. For a free affix non-terminal it is Σ_A^*, where Σ_A^* is the set of all legal affix values. For a defined affix non-terminal a, $L(a)$ is the set of all $t \in AE_T$ such that $a \Rightarrow_{G_{AG}}^* t$. Note that $L(a) = \{a\}$ if $a \in T_A$ and if $a \in S_A$ then $L(a) = \{a_1, a_2, \ldots\}$ where $a_i = a_{i1} \ldots a_{in}$ with $n \geq 0 \wedge \forall_{j \leq n} a_{ij} \in a$.

An *affix assignment* is a function $asg : AE' \rightarrow AE_T$, where $AE' = A'^*$ and $A' \subseteq A$. This function asg is defined as: $asg(\langle x_i \mid 1 \leq i \leq m \rangle) = \langle asg(x_i) \mid 1 \leq i \leq m \rangle$ and

$asg(a) \in L(a)$ for all $a \in A'$. The affix assignment function must satisfy the consistent substitution constraint.

An element $\langle\langle \langle v_i, \langle t_{ij} \mid 1 \leq j \leq m_i \rangle\rangle \mid 0 \leq i \leq m \rangle\rangle$ of $R_{\setminus U_T} \times R^*$ is a *production rule* if there is a hyper rule $\langle\langle \langle v_i, \langle e_{ij} \mid 1 \leq j \leq m_i \rangle\rangle \mid 0 \leq i \leq m \rangle\rangle$ and an affix assignment function asg such that $t_{ij} = asg(e_{ij})$ for $1 \leq j \leq m_i$ and $0 \leq i \leq m$. Let $\mathcal{R}$ be the set of all production rules. The language $L(w)$ of an element $w \in R$ is the set of all $s \in U_T^*$ such that $w \Rightarrow_{\mathcal{R}}^* s$.

For the initial non-terminal $B \in H$ and the input sentence w, we try to find an affix assignment function asg such that the resulting set $\mathcal{R}$ gives: $B \Rightarrow_{\mathcal{R}}^* w$.

Chapter 3

Implementing EAGs

The emphasis in this chapter lies on the construction of the abstract syntax tree along with the affix graph, as well as on the decoration of the affix graph. The combination of both data structures will, in the rest of this thesis, be called a *tree-graph*. In this chapter we will abstract away from incrementality. Thus the tree-graph is completely reconstructed and re-evaluated after each edit action.

We will in a number of steps describe how we can get from an input sentence to a fully decorated tree-graph. The first step to be performed is the recognition of the input sentence. We will use a *left-corner backtrack parser* for this purpose. The reason for using this type of parser is that it can easily be generated and it allows left-recursion in the context-free grammar. Furthermore, this type of parser can easily be combined with the affix value propagation mechanism to obtain affix-directed parsing. We will derive the left-corner backtrack parser, by means of a few transformation steps, from a table-driven deterministic left-corner parser. This will be done in Section 3.1. In Section 3.2 we formalize the structure of the tree-graph and we define a number of access routines on the affix graph nodes. The next step will be the extension of the parser with a mechanism to construct a tree-graph in Section 3.3. In Section 3.4 the last step is described: the decoration of the affix graph nodes with valid affix values.

3.1 Left-corner backtrack parser

A left-corner backtrack parser is a combination of two distinct types of parsers: the left-corner parser and the backtrack parser.

The left-corner properties of a context-free grammar are normally only considered with respect to deterministic parsers. For the definition of deterministic left-corner parsers we refer to [Akk89].

Backtrack parsers are seldom used in actual applications. A detailed description of this type of parser can be found in [AU72]. There are top-down as well as bottom-up backtrack parsers. The former do not allow left-recursion in the grammar, whereas the latter (only) prohibit cycles. Neither type of backtrack parsers has problems with ambiguous context-free grammars. Our main objective is to impose no avoidable restrictions on the specification, not even that the language specified is unambiguous[1]. We will later consider the usefulness and workability of such an ambiguous specification. The introduction of untyped placeholders makes it necessary to use such a powerful parsing mechanism. The price to be paid for using this flexible type of parser is an exponential time complexity.

The combination of backtrack and left-corner parsing was given as an exercise in [AU72]. An elegant description of a left-corner backtrack parser is presented in [Mei86], but this definition is given in 'update schemes' and is beyond the scope of this thesis. A modified version of this parser is also applied in the AGFL-project [Kos91a, Zwo90]. We will provide a more imperative description of this parser, using C-like code.

3.1.1 Left-corner parsing

Left-corner parsing, or LC-parsing, is a combination of both top-down and bottom-up parsing. Given a context-free grammar $G = (N, T, P, S)$. A rule of the form $A : X\alpha. \in P$ is said to have symbol X as its *left-corner*, where X may be either a terminal or a non-terminal. This can be formalized in the notion *left-corner relation*, denoted as $\angle_{lc}$. If $N_0 \in N$ and $N_1 \in (N \cup T)$ the left-corner relation

$$N_1 \angle_{lc} N_0$$

holds, if and only if

$$`N_0 : N_1, \ldots, N_n.\text{'} \in P$$

The reflexive transitive closure of the left-corner relation is denoted as: $\angle_{lc}^*$.

The principle of left-corner parsing is based on the following observations. Suppose the parser must recognize the sentence w given the initial non-terminal S, $S \Rightarrow^* w$.

The parser is always in one of the following situations:

1. The parser is at an arbitrary point in the right hand side of a rule and there are still some members left.

   (a) The next member is a terminal. This terminal must be recognized in the input.

   (b) The next member is a non-terminal A. The parser has to recognize a string in the input which can be derived from this non-terminal. The non-terminal A becomes the current reduction goal, i.e. previous reduction goals are 'forgotten' for the moment. The parser will try to recognize this non-terminal by starting with a rule $B : t\beta. \in P$, where t is the next input symbol and the relation $B \angle_{lc}^* A$ holds.

2. The parser has recognized the right hand side of non-terminal B ($B \Rightarrow^* w_B$). $[A, B]$ denotes that non-terminal A is the current reduction goal and non-terminal B is the current *left-corner symbol*. The relation $B \angle_{lc}^* A$ holds, but there is some work still left to be done before the reduction goal A is satisfied. The parser proceeds with the rule $C : B\gamma. \in P$, for which the relation $C \angle_{lc}^* A$ must hold. The first member in the right

[1]Later on we will show that ambiguity and incremental evaluation do not cooperate smoothly.

hand side of this rule is already recognized and the parser will try to recognize the remaining part γ. In Figure 3.1 the rest of the right hand side γ is tD.

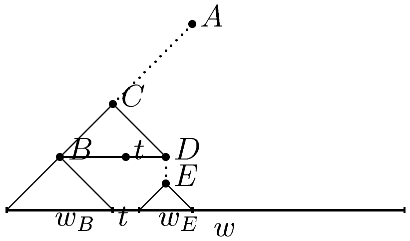


Figure 3.1: Incomplete LC-parse.

3. The parser has recognized the right hand side of non-terminal A and the current reduction goal is non-terminal A, so the current left-corner symbol is the same as the current reduction goal ($[A, A]$). The reduction goal A is satisfied and will be removed as current reduction goal, which makes the previous reduction goal current again.

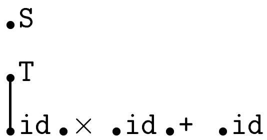
If the reduction goal S (the initial non-terminal) is satisfied and w is completely recognized the parser will report a successful parse.

Given a context-free grammar G_0 with the following rules.

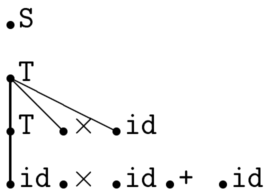
```
S: S,"+",T.            (1)
   T.                  (2)
T: T, "×", id.         (3)
   id.                 (4)
```

The left-corner parse of the sentence `id × id + id` starting with non-terminal `S` is:

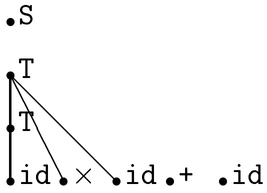
- `S` becomes the current reduction goal, rule (4) fulfills the condition to proceed the parsing process, because $\texttt{T}\angle_{lc}^{*}\texttt{S}$ holds and `id` is the first symbol of the input sentence.



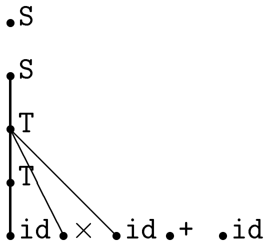
- $[\texttt{S},\texttt{T}]$ holds, the non-terminal `T` is the left-corner of rule (2) and (3), and the relation $\texttt{T}\angle_{lc}^{*}\texttt{S}$ holds. The parser chooses rule (3) (this decision is based on *lookahead*):



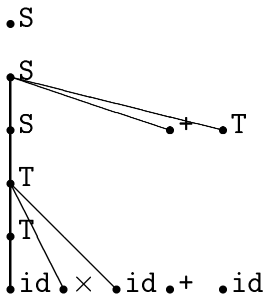
- The parser recognizes the terminals × and `id`:



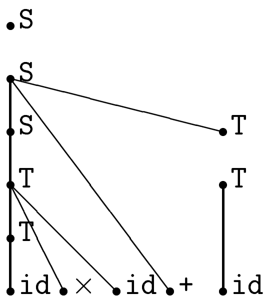
- Again [`S`, `T`] holds and now the parser chooses rule (2):



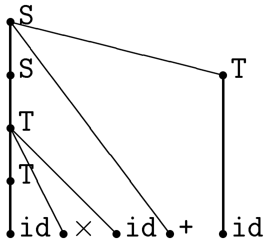
- Although [`S`, `S`] holds the parser is not yet done with parsing since here is still input. `S` is the left-corner of rule (1) and the relation `S`$\angle^*_{lc}$`S` holds so the parser chooses this rule:



- Now the terminal `+` can be recognized. `T` becomes the new current reduction goal. Production rule (4) fulfills the condition to continue the parsing process, because `T`$\angle^*_{lc}$`T` and `id` is the next symbol in the input.



- [`T`, `T`] holds, thus `S` becomes the current reduction goal again, but [`S`, `S`] also holds and there is no input left so:



There are deterministic and non-deterministic LC-parsers. The deterministic LC-parsers are of type LC(k), where $k \geq 0$. The k stands for the number of lookahead symbols necessary to decide which rule the recognized non-terminal is the left-corner of. In the example given above the lookahead was 1 symbol, and this was used, in several cases, to decide which rule was needed to continue parsing. In a LC(0)-grammar each rule has an unique left-corner.

In Exercise 5.1.26 of [AU72] a left-corner parser was introduced. The algorithm given there is a table-driven LC(1)-parser. We shall use this algorithm to explain our version of the left-corner parser, which is a LC(0)-backtrack parser. We will show how, by means of a few small transformation steps, a modified version of this LC(1)-parser can be transformed into our parser.

First, we have to give some definitions. Given is the context-free grammar $G = (N, T, P, S)$. We introduce a *leftmost derivation step*: $wA\delta \Rightarrow_{lm} w\beta\delta$, where $w \in T^*$, $A \in N$, $\delta \in (N \cup T)^*$, and $A : \beta. \in P$. The reflexive transitive closure of this derivation step is denoted as $\Rightarrow^*_{lm}$. If $S = \alpha_0 \Rightarrow_{lm} \alpha_1 \Rightarrow_{lm} \ldots \Rightarrow_{lm} \alpha_n$ is a *leftmost derivation* in G, then we write $S \Rightarrow^*_{lm} \alpha_n$. Such an α_n will be called the *left-sentential form.*
We also have to introduce *left-corner derivation.* A derivation $S \Rightarrow^*_{lc} wA\delta$ is a left-corner derivation if $S \Rightarrow^*_{lm} wA\delta$ is a leftmost derivation. Non-terminal A may not be a left-corner of a rule that introduced this A into a left-sentential form in the derivation $S \Rightarrow^*_{lm} wA\delta$. For example, `S` $\Rightarrow^*_{lc}$ `S+T` is not a legal derivation in G_0, because the left-corner `S` is introduced by the rule `S: S,"+",T`. There are no problems with the derivation `S` $\Rightarrow^*_{lc}$ `a+T` because `T` is not a left-corner of the rule `S: S,"+",T`.
In the algorithm we are going to present we use the notation $\overline{N}$, which means that non-terminal N may produce empty, thus $N \Rightarrow^* \varepsilon$.
We also have to refine the left-corner relation. If $N_0 \in N$ and $M_i \in (N \cup T)$ the left-corner relation

$$M_i \angle_{lc} N_0$$

holds if and only if

$$`N_0 : M_1, \ldots, M_i, \ldots, M_n.\text{'} \in P$$

and $M_1, \ldots, M_{i-1}$ may produce empty.

The LC-parser presented in [AU72] is based on ε-free context-free grammars. The parser is a deterministic parser and uses one symbol lookahead. We shall present an algorithm for generating a parse-table for a non-deterministic version of this parser which uses *no* lookahead. Furthermore, the grammar may contain ε-producing non-terminals. We will use the same notation as in [AU72]. The configuration of the modified LC-parser, denoted by M^T_{LC}, consists of the remaining part of the input and a stack, (w, Γ), where $w \in T^*$,

and $\Gamma \in (N \cup T \cup (N \times N) \cup \varepsilon)^*$.
In the stack element $[A, B]$, which we will call a *reduction marker*, the non-terminal A represents the current reduction goal which is to be recognized and B the recognized left-corner. The parse-table T_{LC} is a mapping from $\Gamma \times (T \cup \varepsilon)$ to $\Gamma^* \cup \{\mathbf{shift}\}$.

If $T_{LC}(X, a) = \beta$ the parser makes the transition $(aw, X\alpha) \vdash_{M^T_{LC}} (aw, \beta\alpha)$.

If $T_{LC}(a, a) = \mathbf{shift}$ it performs the transition $(aw, a\alpha) \vdash_{M^T_{LC}} (w, \alpha)$.

The parser accepts the input sentence if $(w, S) \vdash^*_{M^T_{LC}} (\varepsilon, \varepsilon)$.

Given a context-free grammar $G = (N, T, P, S)$, we will give an algorithm for constructing the parse-table T_{LC}.

1. $B : \alpha. \in P$

   (a) If $\alpha = \epsilon C \beta$ where $C \in N$,
   then $T([A, C], a) = \overline{\epsilon}\beta[A, B]$ for all $A \in N$ and for all $a \in (T \cup \varepsilon)$ such that
   $S \Rightarrow^*_{lc} wA\delta$,
   $A \Rightarrow^* B\gamma$,
   $\epsilon = \epsilon_1 \ldots \epsilon_{i-1}$ where $i > 0 \wedge \forall j < i : \epsilon_j \in N \wedge \epsilon_j \Rightarrow^* \varepsilon$, and
   $\overline{\epsilon} = \overline{\epsilon_1} \ldots \overline{\epsilon_{i-1}}$.
   Note that A may only be the left-corner of some rule if it is S so that A will be a goal at some point in the parsing.

   (b) If $\alpha = \epsilon t \beta$ where $t \in T$,
   then $T(A, a) = \overline{\epsilon} t \beta[A, B]$ for all $A \in N$ and for all $a \in (T \cup \varepsilon)$ such that
   $S \Rightarrow^*_{lc} wA\delta$,
   $A \Rightarrow^* B\gamma$,
   $\epsilon = \epsilon_1 \ldots \epsilon_{i-1}$ where $i > 0 \wedge \forall j < i : \epsilon_j \in N \wedge \epsilon_j \Rightarrow^* \varepsilon$, and
   $\overline{\epsilon} = \overline{\epsilon_1} \ldots \overline{\epsilon_{i-1}}$.

   (c) If $\alpha = \epsilon$ where $\epsilon = \epsilon_1 \ldots \epsilon_i$ where $i \geq 0 \wedge \forall j \leq i : \epsilon_j \in N$,
   then for all $a \in (T \cup \varepsilon)$, $T(B, a) = \overline{B}$ if $i = 0$, or
   $T(B, a) = \overline{\epsilon}$ if $i > 0$.

2. $T(\overline{A}, a) = \varepsilon$ for all $A \in N$, $A \Rightarrow^* \varepsilon$, and for all $a \in (T \cup \varepsilon)$.

3. $T([A, A], a) = \varepsilon$ for all $A \in N$ and for all $a \in (T \cup \varepsilon)$.

4. $T(a, a) = \mathbf{shift}$ for all $a \in T$.

All table entries in the same row are the same for all terminals. It is possible that an entry contains several distinct transitions but M^T_{LC} is non-deterministic and always chooses the appropriate transition.

Consider as an example the context-free grammar G_1 with the following rules.

```
S: "b", U, "e".

U: A.
U: U, "s", A.
```

```
A: M, I.

M: "d".
M: "a".
M: .

I: "i".
```

With this context-free grammar we are able to generate the parse-table T_{LC} of Figure 3.2. The entries for the stack symbols `b`, `e`, `s`, `d`, `a`, and `i` contain the action **shift** which only succeeds if the input symbol corresponds with the stack symbol.

Stack symbol	b, e, s, d, a, i, ε
S	bUe[S, S]
U	d[U, M] $\cup$ a[U, M] $\cup$ i[U, I]
A	d[A, M] $\cup$ a[A, M] $\cup$ i[A, I]
M	d[M, M] $\cup$ a[M, M] $\cup$ $\overline{\mathtt{M}}$
I	i[I, I]
$\overline{\mathtt{M}}$	ε
[S, S]	ε
[U, U]	ε $\cup$ sA[U,U]
[U, A]	[U, U]
[U, M]	I[U, A]
[U, I]	$\overline{\mathtt{M}}$[U, A]
[A, A]	ε
[A, M]	I[A, A]
[A, I]	$\overline{\mathtt{M}}$[A, A]
[M, M]	ε
[I, I]	ε
b	**shift**
e	**shift**
s	**shift**
d	**shift**
a	**shift**
i	**shift**

Figure 3.2: Parse-table T_{LC}.

The initial configuration of the parser is (w, S), where S is the initial non-terminal of the context-free grammar. The final configuration of the parser is $(\varepsilon, \varepsilon)$. For the input sentence `bdisie` the parser M^T_{LC} makes the following transitions:

$$\begin{aligned}(\mathtt{bdisie},\mathtt{S}) &\vdash_{M^T_{LC}} (\mathtt{bdisie},\mathtt{bUe}[\mathtt{S},\mathtt{S}])\\ &\vdash_{M^T_{LC}} (\mathtt{disie},\mathtt{Ue}[\mathtt{S},\mathtt{S}])\end{aligned}$$

$\vdash_{M_{LC}^T}$ (disie,d[U,M]e[S,S])
$\vdash_{M_{LC}^T}$ (isie, [U,M]e[S,S])
$\vdash_{M_{LC}^T}$ (isie,I[U,A]e[S,S])
$\vdash_{M_{LC}^T}$ (isie,i[I,I][U,A]e[S,S])
$\vdash_{M_{LC}^T}$ (sie, [I,I][U,A]e[S,S])
$\vdash_{M_{LC}^T}$ (sie, [U,A]e[S,S])
$\vdash_{M_{LC}^T}$ (sie, [U,U]e[S,S])
$\vdash_{M_{LC}^T}$ (sie,sA[U,U]e[S,S])
$\vdash_{M_{LC}^T}$ (ie,A[U,U]e[S,S])
$\vdash_{M_{LC}^T}$ (ie,i[A,I][U,U]e[S,S])
$\vdash_{M_{LC}^T}$ (e, [A,I][U,U]e[S,S])
$\vdash_{M_{LC}^T}$ (e, $\overline{\texttt{M}}$[A,A][U,U]e[S,S])
$\vdash_{M_{LC}^T}$ (e, [A,A][U,U]e[S,S])
$\vdash_{M_{LC}^T}$ (e, [U,U]e[S,S])
$\vdash_{M_{LC}^T}$ (e,e[S,S])
$\vdash_{M_{LC}^T}$ (ε, [S,S])
$\vdash_{M_{LC}^T}$ $(\varepsilon, \varepsilon)$

The parser M_{LC}^T has two kinds of transitions:

1. If $T_{LC}(a, a) = \textbf{shift}$, then $(aw, a\rho) \vdash_{M_{LC}^T} (w, \rho)$.
2. If $T_{LC}(X, a) = \beta$, then $(aw, X\rho) \vdash_{M_{LC}^T} (aw, \beta\rho)$.

The second element of the machine configuration is a stack containing information used to recognize the remaining part of the input. The top of this stack, along with the next input symbol, is used to select an entry from the parse-table. The derivation of our version of the LC-parser is based on two transformations.

Firstly, the table entries are explicitly coded in the parser. The number of transitions will be equal to the number of entries in the parse-table.

Secondly, we change the contents of the stack from terminals, non-terminals, and reduction markers into procedure calls. A table entry contains either a list of terminals, non-terminals and reduction markers, which should replace the current top of stack, or a symbol indicating that the current top of stack should be replaced by nothing. In the transformed parser the procedure on top of the stack replaces itself by a possibly empty list of procedure calls (each represented by an addresses) (by pushing them one by one onto the stack) and then executes the procedure on top of the stack. Executing a procedure on top of the stack corresponds to popping its address from the stack. If this procedure call stands for an element $t \in T$, then the input pointer may be shifted too. This will be followed by executing the next procedure on top of the stack. We will call this changed stack the *continuation stack*, because it contains all information necessary to continue the parsing process. The resulting parser after performing these transformations will be M_{LC}.

The first M_{LC}^T-transition is not very interesting and can be written as:

1. $(aw, a\rho) \vdash_{M_{LC}} (w, \rho), \forall a \in T$.

The second one is more interesting. It can be split into 5 different transitions:

2a. If $X = n$, where $n \in N$, then
$T_{LC}(n, a) = \overline{\epsilon} a\beta[n, n_0]$
which can be written as:
$(aw, n\rho) \vdash_{M_{LC}} (aw, \overline{\epsilon} a\beta[n, n_0]\rho)$.

2b. If $X = n$, where $n \in N$ and $n \Rightarrow^* \varepsilon$, then
$T_{LC}(n, a) = \overline{\epsilon}$
which can be written as:
$(aw, n\rho) \vdash_{M_{LC}} (aw, \overline{\epsilon}\rho)$.

2c. If $X = \overline{n}$, then
$T_{LC}(\overline{n}, a) = \varepsilon$
which can be written as:
$(aw, \overline{n}\rho) \vdash_{M_{LC}} (aw, \rho)$.

2d. If $X = [n, n]$, where $n \in N$, then
$T_{LC}([n, n], a) = \varepsilon$
which can be written as:
$(aw, [n, n]\rho) \vdash_{M_{LC}} (aw, \rho)$.

2e. If $X = [n, n_i]$, where $n \in N$, then
$T_{LC}([n, n_i], a) = \overline{\epsilon}\beta[n, n_0]$
which can be written as:
$(aw, [n, n_i]\rho) \vdash_{M_{LC}} (aw, \overline{\epsilon}\beta[n, n_0]\rho)$.

For any context-free grammar the set of transitions of M_{LC} can be generated. The set of transitions for the context-free grammar G_1 is:

$(w, \mathtt{S}\rho) \vdash (w, \mathtt{bUe}[\mathtt{S}, \mathtt{S}]\rho)$
$(w, \mathtt{U}\rho) \vdash (w, \mathtt{d}[\mathtt{U}, \mathtt{M}]\rho)$
$(w, \mathtt{U}\rho) \vdash (w, \mathtt{a}[\mathtt{U}, \mathtt{M}]\rho)$
$(w, \mathtt{U}\rho) \vdash (w, \mathtt{i}[\mathtt{I}, \mathtt{I}]\rho)$
$(w, \mathtt{A}\rho) \vdash (w, \mathtt{d}[\mathtt{A}, \mathtt{M}]\rho)$
$(w, \mathtt{A}\rho) \vdash (w, \mathtt{a}[\mathtt{A}, \mathtt{M}]\rho)$
$(w, \mathtt{A}\rho) \vdash (w, \mathtt{i}[\mathtt{I}, \mathtt{I}]\rho)$
$(w, \mathtt{M}\rho) \vdash (w, \mathtt{d}[\mathtt{M}, \mathtt{M}]\rho)$
$(w, \mathtt{M}\rho) \vdash (w, \mathtt{a}[\mathtt{M}, \mathtt{M}]\rho)$
$(w, \mathtt{M}\rho) \vdash (w, \overline{\mathtt{M}}\rho)$
$(w, \mathtt{I}\rho) \vdash (w, \mathtt{i}[\mathtt{I}, \mathtt{I}]\rho)$
$(w, \overline{\mathtt{M}}\rho) \vdash (w, \rho)$
$(w, [\mathtt{S}, \mathtt{S}]\rho) \vdash (w, \rho)$
$(w, [\mathtt{U}, \mathtt{U}]\rho) \vdash (w, \rho)$

$(w, [\mathtt{U}, \mathtt{U}]\rho) \vdash (w, \mathtt{sA}[\mathtt{U}, \mathtt{U}]\rho)$
$(w, [\mathtt{U}, \mathtt{A}]\rho) \vdash (w, [\mathtt{U}, \mathtt{U}]\rho)$
$(w, [\mathtt{U}, \mathtt{M}]\rho) \vdash (w, \mathtt{I}[\mathtt{U}, \mathtt{A}]\rho)$
$(w, [\mathtt{U}, \mathtt{I}]\rho) \vdash (w, \overline{\mathtt{M}}[\mathtt{U}, \mathtt{A}]\rho)$
$(w, [\mathtt{A}, \mathtt{A}]\rho) \vdash (w, \rho)$
$(w, [\mathtt{A}, \mathtt{M}]\rho) \vdash (w, \mathtt{I}[\mathtt{A}, \mathtt{A}]\rho)$
$(w, [\mathtt{A}, \mathtt{I}]\rho) \vdash (w, \overline{\mathtt{M}}[\mathtt{A}, \mathtt{A}]\rho)$
$(w, [\mathtt{M}, \mathtt{M}]\rho) \vdash (w, \rho)$
$(w, [\mathtt{I}, \mathtt{I}]\rho) \vdash (w, \rho)$
$(aw, a\rho) \vdash (w, \rho), \forall a \in T$

A configuration of M_{LC} is of the form (w, ρ) where w is the input sentence, $w \in T^*$, and ρ is a stack, $\rho \in (N \cup T \cup (N \times N) \cup \overline{N})^*$. By considering the stack elements as procedure calls the machine executes the stack itself. In order to make the machine more readable we prefix each stack element with a label indicating which action the parser must perform.

- A stack element $t \in T$ is transformed into S_t.
- A stack element $n \in N$ is transformed into S_n.
- A stack element $[n, m] \in N \times N$ is transformed into $R\_m\ n$, where R_m represents a procedure which either removes the next symbol n from the stack, or pushes a number of procedure calls.
- A stack element $\overline{n}$ is transformed into E_n.

The transitions of the resulting parser M^1_{LC} are:

1. $(tw, S\_t\ \rho) \vdash (w, \rho)$, if $t \in T$ (reading an input symbol).
2. $(w, S\_n\ \rho) \vdash (w, E\_n\ \rho)$, if $n \in N$ and $n \Rightarrow^* \varepsilon$.
3. $(w, E\_n\ \rho) \vdash (w, \rho)$.
4. $(w, S\_n\ \rho) \vdash (w, E\_n_1 \dots E\_n_{i-1}\ S\_t\ S\_n_{i+1} \dots S\_n_k\ R\_n_0\ n\ \rho)$, if $n \in N$ where
   '$n_0 : n_1, \dots, n_{i-1}, t, n_{i+1}, \dots, n_k.$' $\in P$, $\forall j < i : n_j \in N \wedge n_j \Rightarrow^* \varepsilon$, and $t \in T$.
5. $(w, R\_n\ n\ \rho) \vdash (w, \rho)$.
6. $(w, R\_n_i\ n\ \rho) \vdash (w, E\_n_1 \dots E\_n_{i-1}\ S\_n_{i+1} \dots S\_n_k\ R\_n_0\ n\ \rho)$ where
   '$n_0 : n_1, \dots, n_{i-1}, n_i, n_{i+1}, \dots, n_k.$' $\in P$, $\forall j < i : n_j \in N \wedge n_j \Rightarrow^* \varepsilon$.

A configuration of M^1_{LC} is of the form (w, ρ) with
$w \in T^*$ the input sentence, and
$\rho \in (N \cup \{S\_n | n \in (N \cup T)\} \cup \{E\_n | n \in N \wedge n \Rightarrow^* \varepsilon\} \cup \{R\_n | n \in N\})^*$ the continuation stack.

The fourth step of M^1_{LC} is rather complicated and will be split into two smaller steps. We need to introduce a new stack element G_n for each $n \in N$.
So the transition

$$(w, S\_n\ \rho) \vdash (w, E\_n_1 \dots E\_n_{i-1}\ S\_t\ S\_n_{i+1} \dots S\_n_k\ R\_n_0\ n\ \rho)$$

will be split into

$$(w, S\_n\rho) \vdash (w, G\_n\ n)$$

and

$$(w, G\_n\rho) \vdash (w, E\_n_1 \dots E\_n_{i-1} S\_t S\_n_{i+1} \dots S\_n_k R\_n_0 \rho)$$

The transitions of the resulting parser M^2_{LC} are:

1. $(tw, S\_t\ \rho) \vdash (w, \rho)$, if $t \in T$ (reading an input symbol).
2. $(w, S\_n\ \rho) \vdash (w, E\_n\ \rho)$, if $n \in N$ and $n \Rightarrow^* \varepsilon$.
3. $(w, E\_n\ \rho) \vdash (w, \rho)$.
4. $(w, S\_n\ \rho) \vdash (w, G\_n\ n\ \rho)$, if $n \in N$.
5. $(w, G\_n\ \rho) \vdash (w, E\_n_1 \dots E\_n_{i-1}\ S\_t\ S\_n_{i+1} \dots S\_n_k\ R\_n_0\ \rho)$
   where
   '$n_0 : n_1, \dots, n_{i-1}, t, n_{i+1}, \dots, n_k.$' $\in P$, $\forall j < i : n_j \Rightarrow^* \varepsilon$, and $t \in T$.
6. $(w, R\_n\ n\ \rho) \vdash (w, \rho)$.
7. $(w, R\_n_i\ n\ \rho) \vdash (w, E\_n_1 \dots E\_n_{i-1}\ S\_n_{i+1} \dots S\_n_k\ R\_n_0\ n\ \rho)$
   where
   '$n_0 : n_1, \dots, n_{i-1}, n_i, n_{i+1}, \dots, n_k.$' $\in P$, $\forall j < i : n_j \Rightarrow^* \varepsilon$.

The second component of the configuration (w, ρ) of M^2_{LC} is the continuation stack
$\rho \in (N \cup \{S\_n | n \in (N \cup T)\} \cup \{E\_n | n \in N \wedge n \Rightarrow^* \varepsilon\} \cup \{G\_n | n \in N\} \cup \{R\_n | n \in N\})^*$

The generation of the M^2_{LC}-parser is in fact based on implicit grammar transformations. These can also be made explicit, for a detailed description of the explicit grammar transformations see [Ned91]. A non-deterministic LC-parser is useful if the grammar is ambiguous. The non-determinism can be implemented using backtracking, see Section 3.1.2. In Section 3.1.3 we will transform parser M^2_{LC} into C-code.

3.1.2 Backtrack parsers

Backtracking is a general technique for finding solutions to complex problems. One of the most famous problems which can be solved elegantly by using backtracking is the 8-queens problem: put 8 queens on a chessboard in such a way that they cannot take each other.

Backtracking is based on the principle of making one step, seeing whether this step leads to a solution and then undoing this step. If a solution is found, the process may either stop or just report its success and continue (undoing its steps). If in some situation several alternative steps are possible, each of these steps will be tried one after the other.

This strategy can also be used in parsing. The parser either recognizes a symbol or must try some alternative of a rule to recognize the rest of the input sentence. If the parser returns to this situation and there are still alternatives which are not yet tried, these will be tried one after the other. This powerful parsing strategy is not necessary for unambiguous grammars. It can be very helpful if the grammar is ambiguous. Backtracking can be used in both top-down and bottom-up parsers and it can also be combined with the left-corner parsers.

3.1.3 The implementation of the parser

In this section we assume that no (static) semantics is specified, we are only working with the underlying context-free grammar of the eag, which can be described as $G_{CFG} = (N, T, S_H, P, B)$, see Chapter 2,.

In Section 3.1.1 we saw two different ways of implementing left-corner parsers. We gave a number of steps for transforming the table-driven parser into our version of the parser. The parser is not table-driven but the LC-relations are directly included in the code of the parser — for each grammar a complete new parser is generated. In this section we will transform the abstract implementation model into pigeon C-code. Basically we follow the technique given in [Kos75].

The key element of our implementation is the continuation stack, which contains the sequence of text addresses and procedure calls necessary to recognize the remaining part of the input sentence. The stack operations are:

- `push_q(elem)` pushes the element `elem` on the stack, where `elem` is either a text address or procedure call. Pushing a text address is denoted by:

  ```
  push_q("program")
  ```

  Pushing a procedure call will be denoted as:

  ```
  push_q(<<get_program()>>)
  ```

  If the procedure which is pushed has arguments this will be denoted by:

  ```
  push_q(<<sym_symbol("DEFINE")>>)
  ```

  The procedure `push_q` pushes two entities on the stack: the procedure call and its argument.

- `pop_q(n)` pops `n` elements from the stack.

- `top_q()` returns the top element of the stack without popping it from the stack.

We also have the stack operation `call_q()` which pops the top element from the stack and executes the corresponding procedure. If the popped element is not a procedure call the parsing is aborted. We may assume that — due to backtracking — the continuation stack after returning from `call_q()` is in the same state as it was before the execution of the `call_q()`. The structure of the parse-routines is:

```
P()
{
  push_q(M_n);
  ⋮
  push_q(M_1);
  call_q();
  pop_q(n);
}
```

The restoration of the continuation stack after returning from `call_q()` is guaranteed by the fact that each parse-routine implicitly pushes its own address back onto the stack when it is finished. Before a next alternative is tried or the parse-routine stops its execution, the same number of members are popped from the stack by the routine `pop_q()` as were pushed.

Text addresses are used as the reduction goals for the left-corner parser. They represent the non-terminals in the grammar.

The following three parse-routines are generated for each non-terminal N in the grammar:

- `get_`N`()`
- `sym_`N`()`
- `red_`N`()`

If the non-terminal N may produce empty the following parse-routine is also generated:

- `emp_`N`()`

Each of the four types of parse-routines has a specific function, but before presenting them we give some useful definitions.

Definition 1 *An alternative '$N_0 : N_1, \ldots, N_i, \ldots, N_n.$' $\in P$ is called an $\mathtt{LC}^i_T$-alternative of N if $N_0 \angle^*_{lc} N$, $N_j \Rightarrow^* \varepsilon$ $(1 \leq j < i)$ and $N_i \in (T \cup ST)$, where ST is the set of semi-terminals defined in Section 2.2.2.*

Definition 2 *An alternative '$N_0 : N_1, \ldots, N_i, \ldots, N_n.$' $\in P$ is called an $\mathtt{LC}^i_N$-alternative of N if $N_j \Rightarrow^* \varepsilon$ $(1 \leq j < i)$ and $N_i = N$.*

The parse-routine `get_`N`()` is the concrete realization of the continuation stack element G_N in M^2_{LC}. It tries to recognize a prefix of the rest of the input sentence which can be derived from the non-terminal N. If an alternative '$N_0 : N_1, \ldots, N_n$.' $\in P$ is an $\mathtt{LC}^i_T$-alternative of N, then this alternative is included in the parse-routine `get_`N`()`.

The parse-routine `sym_`N is the realization of S_N in M^2_{LC}. As we know from Section 3.1.1, the N may stand for either a terminal or a non-terminal in the parse-routine `sym_`N`()`. If it is a terminal this parse-routine is translated into `sym_symbol(`N`)` which will try to recognize the terminal symbol N. If N represents a non-terminal, this non-terminal is pushed onto the continuation stack as a reduction goal symbol and the parse-routine `get_`N`()` is called. If the non-terminal may produce empty `sym_`N must call the parse-routine `emp_`N`()` as well.

The parse-routine `red_`N`()` is the concrete implementation of R_N in M^2_{LC}. It compares the non-terminal N_T on top of the continuation stack with the non-terminal N; if the two are equal the top of the stack is popped and the procedure on top of the continuation stack is called. The rest of the routine consists of all alternatives which are $\mathtt{LC}^i_N$-alternatives.

The parse-routine `emp_`N`()`, which is the implementation of E_N in M^2_{LC}, is the most simple one. This routine does not affect the input.

3.1.4 Generating an LC-parser

Several aspects of the parser and its generation are demonstrated in this section. For this purpose, we will use the following very simple context-free grammar.

```
program:
  "BEGIN",
    units,
      "END".


units:
  unit.
units:
  units,
    ";",
      unit.

unit:
  application marker,
    identifier.

application marker:
  "DEFINE".
application marker:
  "APPLY".
application marker:
  .

identifier:
  {abcdefghijklmnopqrstuvwxyz} (1).
```

There is no type checking information coded in this simple eag but the following aspects are covered:

- left-recursion,
- empty alternatives, and
- (semi-)terminals.

We will now describe the algorithm for the generation of a parser from an arbitrary eag and illustrate it by the step-by-step generation of a parser for the above example.

1. All non-terminals that may produce empty are marked. In the example grammar there is only one non-terminal that may produce empty: `application marker`. In eags which also describe the static- and dynamic semantics of a language all predicates are marked as empty producing.

2. A routine `sym_`N is generated for each non-terminal N which cannot produce empty. In M^2_{LC} the transition for S_N was:
   $(w, S\_N\ \rho) \vdash (w, G\_N\ N\ \rho)$.
   The following parse-routine is generated for the non-terminal `program`:

```
sym_program()
{
  push_q("program");
  push_q(<<get_program()>>);
  call_q();
  pop_q(2);
}
```

3. The routine `emp_`N is generated for each non-terminal N that may produce empty. The transition for E_N in M^2_{LC} was:
   $(w, E\_N\ \rho) \vdash (w, \rho)$.
   The concrete implementation of this parse-routine for the non-terminal `application marker` is:

```
emp_applicationmarker()
{
  call_q();
}
```

   The routine `sym_`N, for a non-terminal which may produce empty, is more complicated. It is the combined implementation of the following two transitions in M^2_{LC}:
   $(w, S\_N\ \rho) \vdash (w, G\_N\ N\ \rho)$ and
   $(w, S\_N\ \rho) \vdash (w, E\_N\ \rho)$.
   The parse-routine for the non-terminal `application marker`, `sym_application-marker` will be:

```
sym_applicationmarker()
{
  push_q("applicationmarker");
  push_q(<<get_applicationmarker()>>);
  call_q();
```

```
  pop_q(2);
  push_q(<<emp_applicationmarker()>>);
  call_q();
  pop_q(1);
}
```

4. The left-corner relation and its transitive closure are computed. For the example grammar we have:

   "BEGIN" $\angle_{lc}$ program
   "DEFINE" $\angle_{lc}$ application marker
   "APPLY" $\angle_{lc}$ application marker
   application marker $\angle_{lc}$ unit
   identifier $\angle_{lc}$ unit
   unit $\angle_{lc}$ units
   units $\angle_{lc}$ units
   application marker $\angle_{lc}^{*}$ units
   identifier $\angle_{lc}^{*}$ units
   "DEFINE" $\angle_{lc}^{*}$ unit
   "APPLY" $\angle_{lc}^{*}$ unit
   "DEFINE" $\angle_{lc}^{*}$ units
   "APPLY" $\angle_{lc}^{*}$ units

   The alternatives '$N_0 : N_1, \ldots, N_i, \ldots, N_n$.' which are $\mathtt{LC}_T^i$-alternatives of N are collected and used for the generation of the parse-routine get_N. This step of the generation phase corresponds with step (1b) of the algorithm for generating T_{LC}.

   An entry in T_{LC} for a non-terminal N may contain a union of alternatives that can be tried. So in M_{LC}^2 several transitions are also generated for non-terminal N:
   $(w, G\_n\ \rho) \vdash (w, E\_n_1 \ldots E\_n_{i-1}\ S\_t\ S\_n_{i+1} \ldots S\_n_k\ R\_n_0\ \rho)$
   These transitions will be combined in the parse-routine get_N. The parse-routine get_N for the non-terminal units contains the alternatives:

   ```
   unit:  application marker, identifier.
   application marker:  "DEFINE".
   application marker:  "APPLY".
   ```

   because these alternatives are the only $\mathtt{LC}_T^i$-alternatives of units.

   The result will be the following parse-routine:

   ```
   get_units()
   {
     push_q(<<red_unit()>>);
     push_q(<<sym_identifier()>>);
   ```

```
  push_q(<<emp_applicationmarker()>>);
  call_q();
  pop_q(3);
  push_q(<<red_applicationmarker()>>);
  push_q(<<sym_symbol("DEFINE")>>);
  call_q();
  pop_q(2);
  push_q(<<red_applicationmarker()>>);
  push_q(<<sym_symbol("APPLY")>>);
  call_q();
  pop_q(2);
}
```

5. The next phase of the generation process corresponds with step (1a) of the algorithm for generation T_{LC}. This phase is more complicated than the other ones, because several entries of T_{LC} are combined in one parse-routine. For a non-terminal N the entries for the stack symbols $[X, N]$, where X may be any non-terminal, are combined. Each of these entries may contain several alternatives. Therefore in order to implement the parse-routine `red_`N_i we make use of the set of transitions from M^2_{LC}:
   $(w, R\_N_i\ N\ \rho) \vdash (w, E\_N_1 \ldots E\_N_{i-1}\ S\_N_{i+1} \ldots S\_N_k\ R\_N_0\ N\ \rho)$
   Note that these transitions correspond with the $\mathtt{LC}^i_N$-alternatives of N_i; these alternatives are collected and used to generate the parse-routine `red_`N_i.

   Only one $\mathtt{LC}^i_N$-alternative can be found for the non-terminal `application marker`:

   ```
   unit:  application marker, identifier.
   ```

   Before the $\mathtt{LC}^i_N$-alternatives are tried, the transition
   $(w, R\_N\ N\ \rho) \vdash (w, \rho)$
   must be implemented. This is done by a test which checks whether the top of the continuation stack equals the non-terminal N, so whether the reduction goal has already been satisfied.

   The parse-routine will be for the non-terminal `application marker`:

   ```
   red_applicationmarker()
   {
     if (top_q() = "applicationmarker") {
       pop_q(1);
       call_q();
       push_q("applicationmarker");
     };
     push_q(<<red_unit()>>);
     push_q(<<sym_identifier()>>);
     call_q();
     pop_q(2);
   }
   ```

The following `red_units` parse-routine is generated for the left-recursive non-terminal `units`:

```
red_units()
{
  if (top_q() = "units") {
    pop_q(1);
    call_q();
    push_q("units");
  };
  push_q(<<red_units()>>);
  push_q(<<sym_unit()>>);
  push_q(<<sym_symbol(";")>>);
  call_q();
  pop_q(3);
}
```

The alternative

```
units:  units,";",unit.
```

is the only $\mathtt{LC}^i_N$-alternative of `units`.

We have given a typical specimen for each type of parse-routine using the context-free grammar given in the example. A complete parser for an arbitrary context-free grammar can be generated with this algorithm. The complete parser for this example can be found in Appendix B.

This backtrack left-corner parser version uses no lookahead, which in some cases could increase the performance considerably. It will tremendously complicate the generation of the parser.

3.1.5 Parse-routines for predicates

In order to give a full description of the parser we have to take (static) semantics into consideration.

The consequence of extending the grammar with predicates is that we have to extend the left-corner backtrack parser with a new set of parse-routines, to cope with the predicate definitions in the eag. Recall that in the first phase of the parser generator all non-terminals that *only* produce empty and which are therefore predicates, are marked.

If a non-terminal N is the left hand side of a predicate definition the parse-routine `sym_`N has to call the parse-routine `pre_`N. The body of this routine consists of all the alternatives in the right hand side of N.

The parse-routines `sym_`P and `pre_`P for an arbitrary predicate

$$
\begin{aligned}
&P_0 : P_{11}, \ldots, P_{1p_1}.\\
&\quad\vdots\\
&P_0 : P_{n1}, \ldots, P_{np_n}.
\end{aligned}
$$

will look like

```
sym_P0()
{
  push_q(<<pre_P0()>>);
  call_q();
  pop_q(1);
}
```

and

```
pre_P0()
{
  ⋮
  push_q(<<pre_Pip_i()>>);
  ⋮
  push_q(<<pre_Pi1()>>);
  call_q();
  pop_q(p_i);
  ⋮
}
```

Some predicates such as: `pre_equal()` and `pre_notequal()` are primitive. The definitions of the parse-routines for predicates are actually more complicated because of critical affix positions and the possibility of delaying predicates. But we need not consider these aspects for the moment.

The execution of predicates is top-down and the termination of the execution must be ensured via the affix values of the critical affix positions.

3.2 Tree-graph

Before we decorate the affix graph nodes in the tree-graph with affix values, we have to know its structure and how it is constructed. The (incremental) affix evaluation mechanism is based on moving values from one affix graph node to another and (sometimes) performing operations on these values. For ease of presentation we assume, in the rest of this chapter, that the only operator is concatenation. The other operations will be described in Section 5.2.

In this section we discuss the internal structure of the nodes, formalize the structure of the tree-graph, and give a number of access routines for obtaining information from the tree-graph components. This information is used in the description of the (incremental) affix evaluation mechanism.

There are two types of nodes in the tree-graph:

- tree nodes which correspond to hyper non-terminals or hyper sets
- affix graph nodes which correspond to affix non-terminals, affix terminals, and affix sets in affix expressions of displays.

There are also two types of links:

- links between two tree nodes
- links between a tree node and an affix graph node.

A tree node is linked to an affix graph node if and only if the affix non-terminal, affix terminal, or affix set represented by this affix graph node is applied in an affix expression of the display of the corresponding hyper non-terminal or hyper set. Affix non-terminals with the same name within one alternative of a hyper rule are represented by one affix graph node. Affix graph nodes are *never* directly linked to each other.

3.2.1 Tree nodes

There is a unique type of tree node for each alternative in the eag. The node in the tree-graph corresponding to the hyper rule '$N_0(p_1, \ldots p_m) : m_1, \ldots, m_n$.' $\in P$, is represented as:

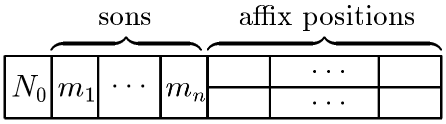


Figure 3.3: Tree node.

As can be seen in Figure 3.3, a tree node consists of an identification which is represented by the non-terminal N_0, links to the subtrees which correspond with the non-terminal and hyper set members in the right hand side of the rule and two rows of affix positions. The affix positions in the upper row represent the affix expressions of non-terminal N_0 in the right hand side of a rule, the affix positions in the lower row represent those in the left hand side. Although the affix expressions are the same for each applying occurrence of the hyper non-terminal, this information is stored because of the operations associated with the affix expressions. The upper and lower side of any one affix position form an *affix position slice*. A different affix expression may be associated with each side of a slice. These expressions may consist of several affix terms. Each affix term is connected to an affix graph node. Therefore each side of an affix position slice may be linked to several affix graph nodes (Figure 3.4).

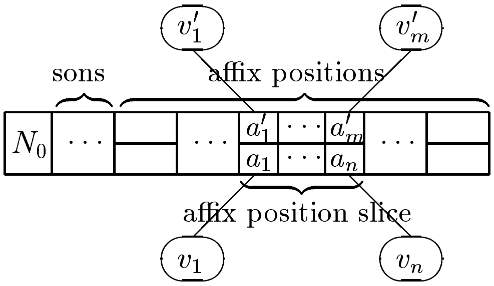


Figure 3.4: Affix position slice.

The value of a side of an affix position slice is obtained by concatenating the values of the connected affix graph nodes, which is only possible if *all* connected nodes have a value. A number, the *sill*, is associated with each side of the affix position slice to give the number of affix graph nodes with no value. Initially this value is equal to the number of affix non-terminals and affix sets in the expression. Each time a value is assigned to an affix graph node the sills of the connected slices are decreased. The sill acts as a semaphore during the evaluation process.

The concatenated values of the affix graph nodes connected to both sides of an affix position slice must be equal.

3.2.2 Affix graph nodes

An affix graph node represents an affix term which is either an affix terminal, an affix non-terminal or an affix set, as we have seen in Chapter 2. If it is an affix terminal then the value of the affix graph node is always a constant value representing this terminal, see Figure 3.5.

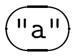


Figure 3.5: Affix graph node for an affix terminal.

If the affix term is a defined affix non-terminal then the value of this affix graph node must be a member of the language defined by the corresponding definition. The affix graph node therefore contains both a value and a function to check this value, see Figure 3.6. If the affix term is an affix set the value must be composed from the elements of the affix set. The affix graph node contains both a value and a function to check this value.

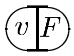


Figure 3.6: Affix graph node for a defined affix non-terminal.

If the affix term is a non-terminal which has no definition the value of the affix graph node is not restricted and the node will contain a function which always yields '`true`'.

As well as a value and a possible function the affix graph node contains at least one link with an affix position slice. These links are established during the parsing process as described in Section 3.3. Each link has one of the following three types:

- undefined (‘`und`’);
- in (‘`in`’);
- out (‘`out`’).

When the affix graph node is connected to some tree node the type of the link is set to ‘undefined’. Immediately after establishing the connection the routine `propagate` is called. This routine will be explained in Section 3.4.3. The routine `propagate` may change the ‘undefined’ type into one of the other two types. These link types are necessary for the incremental evaluation process. In fact by assigning these types to the links we obtain a *dynamic* flow in the affix graph.

3.2.3 A subtree-graph as example

We extend the simple context-free grammar from Section 3.1.4 with type checking information. Each occurrence of an identifier must be defined before it is applied and each identifier may only be defined once. We give not the complete eag but only a few rules.

```
unit (old env, new env):
  application marker (type),
    identifier (id),
      check application (type, id, old env, new env).

application marker ("D"):
  "DEFINE".
application marker ("A"):
  "APPLY".
application marker ("A"):
  .

check application (>"A", >id, >env, env):
  includes (id, env).
check application (>"D", >id, >old env, new env):
  excludes (id, env),
    add (id, old env, new env).

add (>id, >env, "(" + id + "," + env +")"):
  .
```

Although we will present not the construction routines for the tree-graph until Section 3.3, we will now present a small piece of the tree-graph for the input sentence:

```
BEGIN DEFINE i END
```

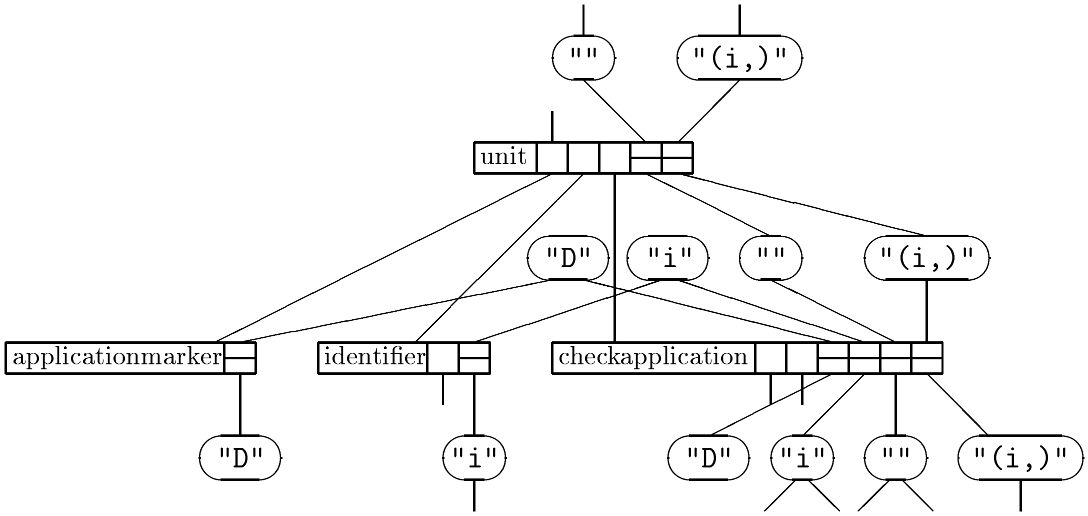


Figure 3.7: Subtree-graph.

The decoration of the affix graph nodes (the ovals) will be discussed in Section 3.4. The types associated with the links between affix graph nodes and affix position slices are omitted in Figure 3.7.

3.2.4 Definition of the graph

In order to discuss the evaluation mechanism we need a more formal description of the tree-graph. We give a number of definitions which are used to define the affix evaluation mechanism.

The set of vertices of a tree-graph TG consists of two disjoint sets:

- the set of vertices $\mathcal{T}$, the tree nodes
- the set of vertices $\mathcal{A}$, the affix graph nodes.

The set of edges of a TG consists of two disjoint sets as well:

- the set of edges $\mathcal{TE}$, representing the links between the tree nodes
- the set of edges $\mathcal{AE}$, representing the links between the affix position slices and the affix graph nodes.

The tree-graph can be split in such a way that we get a (well-defined) tree and a bipartite graph. In order to achieve this we must split each tree node, see Figure 3.8, in set $\mathcal{T}$ in the following way:

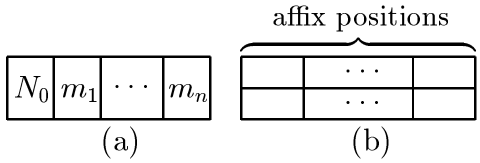


Figure 3.8: Split tree node.

The partitioning of all nodes in set $\mathcal{T}$ results in a set of nodes $\mathcal{P}$, representing the affix position slices, Figure 3.8(b), and a set of nodes $\mathcal{T}'$ representing the tree part of each node of $\mathcal{T}$, Figure 3.8(a). The set of vertices $\mathcal{T}'$ and the set of edges $\mathcal{TE}$ form a tree. The sets of vertices $\mathcal{P}$ and $\mathcal{A}$ together and set of edges $\mathcal{AE}$ form a bipartite graph.

In the description of the bipartite graph we will ignore the underlying tree structure.

A bipartite graph consists of a collection of edges and a collection of two different kinds of vertices. The vertices of the affix graph, the set $\mathcal{A}$, may be considered to be of the first kind and the affix position slices, the set $\mathcal{P}$, of the second. These two sets are disjoint.

$$\mathcal{A} = \{a_1, a_2, \ldots\}$$

$$\mathcal{P} = \{p_1, p_2, \ldots\}$$

The lower side of the affix position slices is represented by 0, whereas the upper side is represented by 1.

An edge in a bipartite graph connects the lower or upper side of a vertex of $\mathcal{P}$ with a vertex of $\mathcal{A}$. The set of edges is $\mathcal{AE}$.

$$\mathcal{AE} = \{e_1, e_2, \ldots\}$$

The functions

$$\texttt{edge\_number\_a} : \mathcal{A} \to \mathbb{N}$$

$$\texttt{edge\_number\_p} : \mathcal{P} \times \{0,1\} \to \mathbb{N}$$

give the number of edges of an affix vertex (Figure 3.9) and a lower or upper side of an affix position vertex (Figure 3.10) respectively.

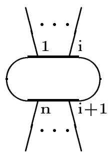


Figure 3.9: Affix graph node with `n` edges.

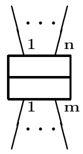


Figure 3.10: Affix position slice with, `n` upper edges and `m` lower edges.

For a vertex $a \in \mathcal{A}$ and a position $i \in \mathbb{N}$, $\texttt{edge\_a(}a\texttt{,}i\texttt{)}$ gives the edge $e \in \mathcal{AE}$, which is connected to the vertex a at position i.

$$\texttt{edge\_a} : \mathcal{A} \times \mathbb{N} \to \mathcal{AE}$$

For a vertex $p \in \mathcal{P}$, a side $j \in \{0,1\}$ and a position $i \in \mathbb{N}$, $\texttt{edge\_p(}p\texttt{,}j\texttt{,}i\texttt{)}$ gives the edge $e \in \mathcal{AE}$, that is connected to the vertex p at position i of side j.

$$\texttt{edge\_p} : \mathcal{P} \times \{0,1\} \times \mathbb{N} \to \mathcal{AE}$$

For an edge $e \in \mathcal{AE}$ the function $\texttt{vertex\_a}(e)$ gives the vertex $a \in \mathcal{A}$ that is connected to the edge e at the affix graph node side and the position where the edge is connected to the vertex.

$$\texttt{vertex\_a} : \mathcal{AE} \rightarrow \mathcal{A} \times \mathbb{N}$$

The function $\texttt{vertex\_p}(e)$ gives the vertex $p \in \mathcal{P}$ that is connected to the edge e at the affix position side, a number to indicate the side of the slice and the position where the edge is connected to the vertex.

$$\texttt{vertex\_p} : \mathcal{AE} \rightarrow \mathcal{P} \times \{0,1\} \times \mathbb{N}$$

We now define some extra functions which will be needed by the affix evaluation mechanism. The function

$$\texttt{type\_of} : \mathcal{AE} \rightarrow \{\text{`}\texttt{und}\text{'}, \text{`}\texttt{out}\text{'}, \text{`}\texttt{in}\text{'}\}$$

gives the type of a link in the bipartite graph.

The function

$$\texttt{value\_of} : \mathcal{A} \rightarrow \Sigma_A^* \cup \{\perp\}$$

gives the value of an affix graph vertex. If the affix graph node has no value yet the value $\perp$ is returned. Σ_A^* is the alphabet of the legal affix values. If the vertex does not yet have a value this function returns the 'undefined' value. The function

$$\texttt{has\_value} : \mathcal{A} \rightarrow \{0,1\}$$

which checks whether the affix graph node has some value can be derived from the previous function in the following manner:

$$\texttt{has\_value}(e) = \begin{cases} 0 & \text{if } \texttt{value\_of}(e) = \perp \\ 1 & \text{if } \texttt{value\_of}(e) \neq \perp \end{cases}$$

The function

$$\texttt{affix\_of} : \mathcal{P} \times \{0,1\} \times \mathbb{N} \rightarrow \mathcal{A}$$

gives the affix graph node connected to a side of an affix position slice at position j. It is defined as:

$$\begin{array}{l} \texttt{affix\_of}(p,i,j) = a \\ \text{where} \\ (a,n) = \texttt{vertex\_a}(\texttt{edge\_p}(p,i,j)) \end{array}$$

We mentioned, in Section 3.2.1, that a sill is associated with each side of an affix position slice. We are now able to define this sill more formally.

$$\texttt{sill} : \mathcal{P} \times \{0,1\} \rightarrow \mathbb{N}$$

This function gives the number of affix graph nodes which do not yet have a value. Using the function definitions above this sill function may be defined as follows.

$$\texttt{sill}(p,i) = \texttt{edge\_number\_p}(p,i) - \sum_{j=1}^{\texttt{edge\_number\_p}(p,i)} \texttt{has\_value}(\texttt{affix\_of}(p,i,j))$$

The last function which must be defined is associated with the definitions of an affix non-terminal or an affix set. Each affix graph node representing a non-terminal or set contains a function. The function

$$\texttt{affix\_function\_of} : \mathcal{A} \rightarrow (\Sigma_A^* \rightarrow \{\texttt{true}, \texttt{false}\})$$

returns the function which checks the value against the definition associated with the affix non-terminal or affix set. If the affix non-terminal is free the function delivers which always '`true`' is returned.

3.3 Tree-graph construction

The parser must not only recognize the input sentence, but also construct the corresponding tree-graph. In this section we describe the adaptations in the parse-routines necessary to construct the tree-graph.

Given an eag with the underlying context-free grammar
$G_{CFG} = (N, T, S_H, P, B)$;
consider a hyper rule:
'$N_0 : N_1, \ldots, N_m.$' $\in P$ where
$N_0 \in H$ and $N_1, \ldots, N_m \in (H \cup T \cup S_H)$. Suppose that a node has to be created in the tree-graph for a non-terminal. Such a node can be created during the recognition of the rule[2]

1. before the parser starts to recognize the right hand side.
2. upon completion of the recognition of the right hand side.
3. at any moment between starting and finishing the recognition, for example, after the first member has been recognized.

A lot of unnecessary work may be done in the first case if the first member can not be recognized, since our parsers do not use lookahead.

The second solution restricts the principle of affix-directed parsing. Whether the recognition of two members of a right hand side would lead to an inconsistent affix graph could only be detected after the entire right hand side is recognized.

The third solution is a compromise between the other two. A node is created when the parser has recognized the first member. This approach also meshes perfectly with left-corner parsing.

3.3.1 The algorithm

We will demonstrate the algorithm for constructing the tree-graph by means of an arbitrary alternative in an eag.

[2]Recognition of a rule means that the parser tries to recognize a part of the input derivable from the right hand side of the rule.

- The algorithm starts with the creation of a number of affix graph nodes necessary in this alternative, these nodes are only locally available.

- The parser will either try to recognize the left-corner of this alternative, which will always be a (semi-)terminal, or it has just recognized the left-corner of this alternative, which will always be a non-terminal. In the first case the parser is in a `get_`*N* parse-routine and in the second case in a `red_`*N* parse-routine. If the alternative starts with possible empty producing members and the parse-routine `get_`*N* is executed, then the tree node is created before these possible empty producing members are recognized. If the alternative does not start with possible empty producing members and the parse-routine `get_`*N* is executed, then the tree node is created immediately after having recognized the first member. If the parse-routine `red_`*N* is executed, then the tree node is created immediately after having recognized the left-corner.

- The created tree node is pushed onto a separate tree node stack. If the alternative is recognized by way of a `red_`*N* parse-routine the subtree built for left-corner non-terminal is immediately linked to created tree node.

- The parser proceeds with the recognition of the rest of the alternative.

  – Non-terminals and hyper sets are recognized and the root node for the constructed subtree-graphs are on top of the tree node stack. These root nodes are removed and linked to the next element on the tree node stack, which is the tree node 'under construction' for this alternative. The subgraph is also connected to the relevant affix graph nodes.

  – Terminals are recognized and no nodes are created for them.

- The parser also has a backtrack phase, during this phase all links and nodes created are recursively demolished.

3.3.2 Construction routines

There are 4 routines involved in the creation of the tree-graph.

`make_tree_node`: for the creation of tree nodes.

`make_affix_graph_node`: for the creation of affix graph nodes.

`make_tree_link`: for creating links between tree nodes.

`make_affix_link`: for creating links between tree nodes and affix graph nodes.

The calls of the routines `make_tree_node`, `make_tree_link`, and `make_affix _link` in the parse-routines `get_`*N* and `red_`*N* are pushed onto the continuation stack, in order to create the tree node, tree link, and affix link respectively. In this way the creation of tree nodes and links is fully integrated in the backtrack mechanism. The three routines show an equal respect of the backtrack mechanism — all allocated memory is freed during backtracking, no traces of the nodes or links are left behind.

The addresses of the tree nodes created by `make_tree_node` are pushed onto a separate tree node stack. The addresses remain on this stack as long as the nodes are not completed. A node is completed if all links between it and its sons are created and the node is also connected to all related affix graph nodes. The tree node stack corresponds to the path in the tree-graph from the node under construction to the top. As soon as the link between the tree node N and its father F is made by the routine `make_tree_link` the address of N is removed from the stack and the tree node N is then considered to be completed. The father node F is always the next node on the tree node stack.

The routine `make_affix_link(`A`,`S`)` creates a link between an affix graph node A and an affix position slice S of the tree node on top of tree node stack. We assume that each affix position has an implicit name, which makes it possible to refer to a specific affix position slice of a tree node.

The call of the routine `make_affix_graph_node` is not pushed onto the continuation stack. Before the recognition of an alternative starts the affix graph nodes used within the alternative are created. The backtrack mechanism requires that these nodes be explicitly dismantled during backtracking and the routine `free_affix_graph_node` is therefore also defined. The explicit creation of affix graph nodes offers us the possibility of assigning names to them, which is quite useful because several distinct tree nodes may be connected to the same affix graph node.

3.3.3 An example

In Section 3.2.3 we gave a few hyper rules which will be used for generating new parse-routines. The parse-routine `get_units`, for example, will be:

```
get_units()
{
  {
    oldenv = make_affix_graph_node();
    ⋮
    id = make_affix_graph_node();
    push_q(<<red_unit()>>);
    ⋮
    push_q(<<sym_identifier()>>);
    push_q(<<make_tree_link()>>);
    push_q(<<make_affix_link(type,pos1)>>);
    push_q(<<emp_applicationmarker()>>);
    push_q(<<make_affix_link(newenv,pos2)>>);
    push_q(<<make_affix_link(oldenv,pos1)>>);
    push_q(<<make_tree_node("unit")>>);
    call_q();
    pop_q(16);
```

```
    free_affix_graph_node(id);
    ⋮
    free_affix_graph_node(oldenv);
  };
  {
    loc_1 = make_affix_graph_node("D");
    push_q(<<red_applicationmarker()>>);
    push_q(<<make_affix_link(loc_1,pos1)>>);
    push_q(<<make_tree_node("applicationmarker")>>);
    push_q(<<sym_symbol("DEFINE")>>);
    call_q();
    pop_q(4);
    free_affix_graph_node(loc_1);
  };
  ⋮
}
```

Of course a lot of extra information will necessary for the creation of the nodes, but for sake of the example we want to avoid too much detail. We also assume for the sake of simplicity that all routines are polymorphic.

3.3.4 Optimalizations

Since a tree node is created for each alternative of a rule the tree-graph will have the same size as the derivation tree. The tree-graph thus built is not really 'abstract'. In the SSL [RT89b] the specification writer has to indicate where nodes of the abstract syntax tree must be created. In our system the tree-graph is built automatically.

Two heuristic rules can be formulated to obtain a more efficient tree-graph.

Rule 1 A tree node is not created for an alternative of a rule of which the right hand side consists of exactly one non-terminal member and in which no operations are performed on the affixes in either the left or right hand sides. Applying this rule will eliminate chain productions in the tree-graph.

Rule 2 The subtrees constructed for left- or right-recursive rules can be flattened if the rule has the following pattern:

```
A (old affix, new affix):
  B (old affix, affix),
    "terminal",
      A (affix, new affix).
A (old affix, new affix):
  B (old affix, new affix).
```

and no operations are performed on the affixes in these alternatives. A subtree of the following structure will normally be built for a right-recursive rule:

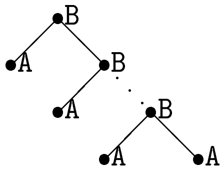


Figure 3.11: Subtree for right-recursive rule.

The flattened subtree will look like:

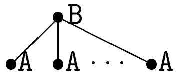


Figure 3.12: Flattened subtree.

The application of this rule will have a major impact on the other parts of the system which use the tree-graph such as the unparser.

In the ASF+SDF meta-environment [Kli91] similar rules are formulated to obtain an optimal abstract syntax tree [HHKR89]. These rules are as yet not implemented in the prototype of PREGMATIC, but are in development.

3.4 Non-incremental affix evaluation

Affix evaluation and affix value propagation are two names for the same phenomenon, viz. assigning values to the nodes in the tree-graph. This will be the last step in the process of transforming an input sentence into a fully decorated tree-graph.

Incremental affix evaluation consists of two parts: the propagation of affix values and the propagation of 'undefined'. The latter will be explained in Section 5.2. In this section we concentrate on the propagation of affix values.

A tree-graph consists of two types of nodes. The propagation algorithm consists of two parts. One part takes care of the propagation of a value from an affix graph node to a tree node and the other part from a tree node to an affix graph node. This distinction may seem artificial, but propagation can start in both an affix graph node and a tree node.

Given an affix graph node with a value, this value is propagated to one of the linked affix position slices. The value is moved from side s of an affix position slice to the other side s' and is then propagated to affix graph nodes connected to this side. Note that, on both sides of the slice, operations may be performed on the value. On side s other values may be concatenated and on side s' the value may be split into several parts, which are all propagated to distinct affix graph nodes. This process is continued, possibly also with values from other affix graph nodes, until either all affix graph nodes have a value or until an attempt is made to assign a value to an affix graph node which either already contains a value which is not the same as the propagated one, or whose associated definition does not accept the propagated value.

This informal description of our propagation algorithm resembles the algorithms used in systems based on attribute grammars. The evaluation process in systems based on attribute grammars usually starts assigning values to the nodes in the attribute graph after completing the abstract syntax tree and the associated attribute graph. Several algorithms have been developed to determine the order in which the attributes have to be visited in order to calculate the values of the attributes in the graph [Alb89c, Alb89b, Alb89a, LMOW88, RTD83]. In contrast our propagation mechanism is activated as soon as a link is created between an affix graph node and an affix position slice. It is not necessary for the tree-graph to be complete, the evaluation mechanism will try to assign values to affix graph nodes as soon as possible.

The parsing process and the affix value propagation are intermixed to obtain affix-directed parsing. The affix value propagation mechanism is also based on backtracking. The mechanism propagates values depth-first through the affix graph in an eager way. Suppose a node A in this graph has several undefined links. The process selects one of these links and starts to propagate the value of this node. If this process stops somewhere in the graph, the next undefined link is selected in node A and the propagation is started again. This is repeated until there are no undefined links left. This process works recursively in all visited nodes.

For all affix graph vertices in the bipartite graph the following condition holds:

$$\forall a \in \mathcal{A} : \texttt{value\_of}(a) = \perp \quad \vee \quad \begin{array}{l} \texttt{F}(\texttt{value\_of}(a)) = \text{`}\texttt{true}\text{'} \\ \text{where} \\ \texttt{F} = \texttt{affix\_function\_of}(a) \end{array}$$

This condition must always be satisfied during propagation.

3.4.1 Propagation from affix graph node to tree node

The affix value propagation mechanism can be split into two phases. The first phase takes care of the propagation from an affix graph node to a tree node. The requirements for starting the propagation process in an affix graph node A are: the node must have a value and at least one undefined link. Thus, if the condition

$$\texttt{has\_value}(A) = 1 \wedge (\exists j : 1 \leq j \leq \texttt{edge\_number\_a}(A) \wedge \texttt{type\_of}(\texttt{edge\_a}(A,j)) = \text{`}\texttt{und}\text{'})$$

holds, the routine `propagate` will continue.

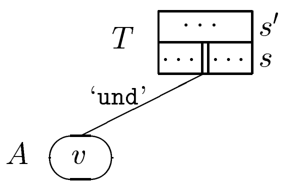


Figure 3.13: Subgraph with undefined link.

The selected link refers to an affix position slice side which may be connected to several other affix graph nodes. The type of the link between A and T becomes 'out', to indicate that the affix value has 'left' this node by this link, and the sill of the corresponding side s of the affix position slice is decreased by one.

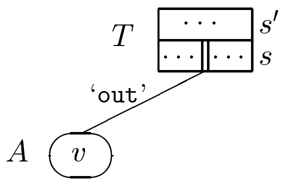


Figure 3.14: Subgraph with 'out' link.

If `sill(`T`,`s`) = 0`, which indicates that *all* affix graph nodes corresponding to the affix terms in the affix expression have a value, the values are concatenated and the result value is propagated to the other side s' of the slice. The action performed by the routine `propagate` depends on the `sill(`T`,`s'`)`, as will be described in the next section.

3.4.2 Propagation from tree node to affix graph node

This is the second phase in the propagation mechanism. If a value v is propagated from side s to side s' the actions performed by the routine `propagate` depend, first of all, on the completeness of the affix graph at that point.

- In the first case, some parts of the tree-graph are not known when propagation starts. For example, the other side of an affix position slice may not as yet be connected to affix graph nodes. In such a situation the propagation stops in this node. It will be resumed later when this affix position slice is linked to one or more affix graph nodes.

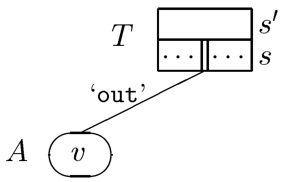


Figure 3.15: Incomplete subgraph.

The condition that has to be fulfilled before the value can be propagated is:

$$\texttt{sill}(T,s) = \texttt{0} \wedge \texttt{edge\_number\_p}(T,s') > \texttt{0}$$

- Otherwise, the other side of the affix position slice is connected to one or more affix graph nodes.

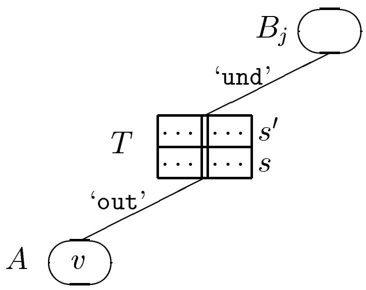


Figure 3.16: Complete subgraph.

The propagated value

$$v_1 + \ldots + v_{\texttt{edge\_number\_p}(T,s)}$$

is non-deterministically split into $\texttt{edge\_number\_p}(T,s')$ parts. This means that the propagation mechanism has to generate all the splits of the propagated value that satisfy the constraints imposed by the nodes connected to side s'.

The following condition holds for all links between affix position slice T and affix graph nodes B_j.

$$\forall_{1 \le j \le \texttt{edge\_number\_p}(T,s')} :$$

$$\texttt{type\_of}(\texttt{edge\_p}(T,s',j)) = \text{`}\texttt{und}\text{'} \vee \texttt{type\_of}(\texttt{edge\_p}(T,s',j)) = \text{`}\texttt{out}\text{'}$$

– Some of the links between T and nodes B_j may have the type 'out'. Such affix graph nodes must already have a value. The part of value that has to be propagated to such a node must be equal to the present value of the node, in order to satisfy the consistent substitution constraint.

  If the condition

$$\texttt{has\_value}(B_j) = \texttt{1}$$

  holds, then `propagate` compares the value of B_j with the propagated value. Let

$$z = \sum_{i=1}^{\texttt{edge\_number\_p}(T,s)} \texttt{value\_of}(\texttt{affix\_of}(T,s,i))$$

  be the propagated value. Then we should have

$$\exists u, v : \; u \; + \; \texttt{value\_of}(B_j) \; + \; v \; = \; z.$$

– The other possible type of link is 'undefined'. The action of the propagation mechanism depends on the value of node B_j.

  If the node already has some value, the same situation arises as above. If the values are equal the link between T and B_j becomes 'in'.

  Otherwise the node however has no value. Let

$$z = \sum_{i=1}^{\texttt{edge\_number\_p}(T,s)} \texttt{value\_of}(\texttt{affix\_of}(T,s,i))$$

the propagated value. In order to assign a part of the value to the affix graph node B_j this value must satisfy the condition

$$\exists u_1, \ldots, u_{\texttt{edge\_number\_p}(T,s')} :$$

$$\sum_{i=1}^{j-1} u_i \; + \; u_j \; + \; \sum_{i=j+1}^{\texttt{edge\_number\_p}(T,s')} u_i \; = \; z$$
$$\wedge$$
$$F(u_j) = \text{‘}\texttt{true}\text{’}$$
where
$$F = \texttt{affix\_function\_of}(B_j)$$

If the propagated value satisfies this condition the type of link between T and B_j becomes ‘in’. The propagation mechanism will recursively propagate the value via the ‘undefined’ links of B_j, if any.

If the propagated value does not satisfy the constraints, the value is not assigned to the node B_j and the propagation starts to backtrack, because an inconsistency was detected in the affix graph, i.e. some context condition was violated.

Whenever the sills of both sides of the affix position slice are zero the following condition *is guaranteed to* hold:

$$\sum_{j=1}^{\texttt{edge\_number\_p}(T,s)} \texttt{value\_of}(\texttt{affix\_of}(T,s,j))$$
$$=$$
$$\sum_{j=1}^{\texttt{edge\_number\_p}(T,s')} \texttt{value\_of}(\texttt{affix\_of}(T,s',j))$$

3.4.3 Propagation algorithms

In Sections 3.4.1 and 3.4.2 we discussed the principles of, and conditions for the propagation of affix values. In this section we will give the algorithms themselves. We first give the algorithm for propagating values from affix position slices to affix graph nodes.

```
propagate′(edge) /* from tree node to graph node*/
{
 (slice,side,pos) := vertex_p(edge);
 sill(slice,side) -:= 1;
 type_of(edge) := ‘out’;
 if (sill(slice,side) = 0) {
  side′:= 1 - side;
  nr := edge_number_p(slice,side′);
  Σ_{i=1}^{nr} val_i := Σ_{j=1}^{edge_number_p(slice,side)} value_of(affix_of(slice,side,j));
  for i := 1 to nr
   push_q(<<propagate(slice,side′,i,val_i)>>);
  call_q();
  pop_q(nr);
 }
```

```
 else
  call_q();
 type_of(edge) := ‘und’;
 sill(slice,side) +:= 1;
}
```

This routine propagate′ is quite simple, it simply splits the value non-deterministically and propagates the values $\mathtt{val_i}$ through the links, independently of the types of those links. The expression `1 - side` switches the process from the lower or upper affix position slice side to the upper or lower side respectively.

We now give the algorithm for propagating values from affix graph nodes to affix position slices.

```
propagate(slice,side,nr,value) /* from graph node to tree node*/
{
 node := affix_of(slice,side,nr);
 edge := edge_p(slice,side,nr);
 if (value_of(node) <> ⊥) {
  if (value_of(node) = value)
   if (type_of(edge) = ‘und’) {
    sill(slice,side) -:= 1;
    type_of(edge) := ‘in’;
    call_q();
    type_of(edge) := ‘und’;
    sill(slice,side) +:= 1;
   }
   else
    call_q();
  }
  else /*backtrack*/
 }
 else {
  F := affix_function_of(node);
  pushed := 0;
  if (F(value)) {
   value_of(node) := value;
   sill(slice,side) -:= 1;
   type_of(edge) := ‘in’;
   for i := 1 to edge_number_a(node) {
    edge′ := edge_a(node,i);
    if (type_of(edge′) = ‘und’) {
     pushed +:= 1;
     push_q(<<propagate′(edge′)>>);
    }
   };
   call_q();
```

```
   pop_q(pushed);
   type_of(edge) := 'und';
   sill(slice,side) +:= 1;
  }
 }
 else /*backtrack*/
}
```

Depth-first behaviour is obtained by processing a `propagate`- or `propagate`′-call completely before the next `propagate` or `propagate`′ is popped from the stack.

The else parts marked by the comment `/*backtrack*/` invoke the backtrack mechanism. This mechanism can be implemented is several ways, for example that described in [Mei92].

3.4.4 Predicates and propagation

The parser in a programming environment generated uses affix-directed parsing. Context conditions are explicitly checked by predicates. Affix-directed parsing is obtained by evaluating these predicates during the recognition of an input sentence. This makes it necessary for the affix evaluation mechanism to trigger the execution of the predicates. A predicate may be evaluated if the sills of *all* its critical affix position slices are zero, viz. the affix graph nodes connected to the upper side of *all* critical affix position slices have a value. A predicate remains delayed until this is the case.

The delaying mechanism of predicates uses a few extra access routines. Some adaptations to the critical affix position slices also are necessary. A critical affix position slice contains, in addition to the two rows of links to affix graph nodes, a marking that indicates that this affix position slice is critical and a routine call to the delayed predicate with its arguments, see Figure 3.17.

c	`delay_`$P(pos_1,\ldots,pos_n)$		...	
	...			

Figure 3.17: Critical affix position slice.

We need the function:

$$\texttt{is\_critical} : \mathcal{P} \rightarrow \{\texttt{true}, \texttt{false}\}$$

which checks whether an affix position slice is marked as critical, and we need a function `delayed_function` which retrieves the stored delayed routine in a critical affix position slice. Non-critical affix position slices are implicitly marked.

In Section 3.1.5 the body of the parse-routine `pre_`P for the predicate P was presented. This parse-routine is in reality more complicated, as we shall see in this section. We will demonstrate `pre_`P by means of the hyper rules given in Section 3.2.3. The parse-routine `pre_checkapplication` will be of the form:

```
pre_checkapplication()
{
  push_q(<<mark_as_critical(pos1,pos2,pos3,
          <<delay_checkapplication(pos1,pos2,pos3,pos4)>>)>>);
  push_q(<<make_tree_node("checkapplication")>>);
  call_q();
  pop_q(2);
}
```

The routine `mark_as_critical()` takes care of the initialization of the administration, it stores a function call `delay_`P in each critical affix position slice and marks each such position as critical.

The parse-routine `delay_checkapplication` will be of the form:

```
delay_checkapplication(pos1,pos2,pos3,pos4)
{
  if (sill(pos1) = 0 ∧ sill(pos2) = 0 ∧ sill(pos3) = 0) {
    {
      env = make_affix_graph_node();
      loc_1 = make_affix_graph_node("A");
      id = make_affix_graph_node();
      push_q(<<make_affix_link(env,pos4)>>);
      push_q(<<make_tree_link()>>);
      push_q(<<make_affix_link(env,pos2)>>);
      push_q(<<make_affix_link(id,pos1)>>);
      push_q(<<pre_includes()>>);
      push_q(<<make_affix_link(env,pos3)>>);
      push_q(<<make_affix_link(id,pos2)>>);
      push_q(<<make_affix_link(loc_1,pos1)>>);
      call_q();
      pop_q(8);
      free_affix_graph_node(id);
      free_affix_graph_node(loc_1);
      free_affix_graph_node(env);
    };
    ⋮
  }
  else
    call_q();
}
```

Before executing a predicate the affix evaluation mechanism checks whether the affix graph nodes connected to the upper side of *all* critical affix position slices of this predicate have a value. If these affix graph nodes are all defined the predicate is executed. If, however, some of these affix graph nodes do not yet have a value, the predicate is not yet executed. The evaluation of predicates has as side effect that affix graph nodes connected to the non-critical affix position slices get a value.

In order to trigger the execution of a delayed predicate the propagation algorithm propagate′ is adapted.

```
propagate′(edge) /* from tree node to graph node*/
{
 (slice,side,pos) := vertex_p(edge);
 sill(slice,side) -:= 1;
 type_of(edge) := ‘out’;
 if (sill(slice,side) = 0) {
  if (is_critical(slice)) {
   P := delayed_function(slice);
   push_q(<<P>>);
   call_q();
   pop_q(1);
  }
  else {
   side′:= 1 - side;
   nr := edge_number_p(slice,side′);
   ∑_{i=1}^{nr} val_i := ∑_{j=1}^{edge_number_p(slice,side)} value_of(affix_of(slice,side,j));
   for i := 1 to nr
    push_q(<<propagate(slice,side′,i,val_i)>>);
   call_q();
   pop_q(nr);
  }
 }
 else
  call_q();
 type_of(edge) := ‘und’;
 sill(slice,side) +:= 1;
}
```

3.4.5 Cycles and propagation

In Section 2.2.4 we also used the critical affix positions to ensure the termination of cycles during parsing. The eag for the Pascal expressions presented in Section 2.3 is an example of a non-well-formed eag. This is caused by the alternative:

```
term(prio):
  term(prio+1).
```

The corresponding `red_term()` parse-routine, without tree-graph construction routine-calls and affix evaluation, looks like:

```
red_term()
{
  if (top_q() = "term") {
    pop_q(1);
    call_q();
    push_q("term");
  };
  push_q(<<red_term()>>);
  call_q();
  pop_q(1);
}
```

Without affix evaluation this clearly results in non-termination of the parsing process. The parser generator will determine which alternatives are members of a cycle and if there are no critical affix expressions associated with the non-terminal in the left hand side, the eag will be regarded as not being well-formed and is rejected. The alternative of `term` should therefore be specified as:

```
term(>prio):
  term(prio+1).
```

The corresponding `red_term()` routine will then look like:

```
red_term()
{
  if (top_q() = "term") {
    pop_q(1);
    call_q();
    push_q("term");
  };
  push_q(<<mark_critical_affix_positions(pos1,<<delay_red_term(pos1)>>)>>);
  call_q();
  pop_q(1);
}
```

The parse routine `delay_red_term()` checks whether another tree node should be created, if this is not necessary the cycle is considered as completed. The routine `delay_red_term()` looks like:

```
delay_red_term(pos1)
{
  if (sill(pos1) = 0) {
    val_u := Σ_{j=1}^{edge_number_p(slice,side)} value_of(affix_of(slice,side,j));
    val_l := Σ_{j=1}^{edge_number_p(slice,side)} value_of(affix_of(slice,side,j));
    if (val_u = val_l)
      call_q();
```

```
    else {
      if (val_u ∈ prio  ∧  val_l ∈ prio) {
        push_q(<<delay_red_term(pos1)>>);
        /* routine-calls for creating a tree node and
            inserting it in the tree-graph */
        call_q();
        pop_q(i);
      }
      else /* backtrack */
    }
  }
  else
    call_q();
}
```

3.4.6 Propagation and defined non-terminals

In Section 2.2.3 we described a few problems concerning defined affix non-terminals. Affix graph nodes representing these affix non-terminals should be treated in a special way.

After the input sentence is recognized and affix value propagation stops it is still possible that there may be a number of affix graph nodes without a value. This is caused by generative defined affix non-terminals. All affix graph nodes containing a defined affix non-terminal or a finite affix set are checked, those without a value and containing a defined affix non-terminal describing a finite language or a finite affix set start to generate all possible values which are propagated one by one. This process is repeated until either no affix graph nodes, or only affix graph nodes containing defined affix non-terminals describing an infinite language are left. In the latter case it may be impossible to find a fully decorated tree-graph in finite time. In that case the process stops and *no* successful parse is reported.

Defined affix non-terminals describing infinite languages will have only a recognizing function within this strategy.

Chapter 4

Structure of generated environments

In this chapter we give a description of the user-interface and the unparser, which are both non-incremental. We also look at the derivation of language dependent features of the interface, such as the placeholders and the templates.

4.1 User-interface

The user-interface is based on the X-window system and has been implemented using the OLIT-widget set [Sun90a, Sun90b]. It consists of one main window (Figure 4.1) and two windows which pop up if they are needed by the user: a text edit window and a layout modification window. The main window consists of 3 parts: the focus window, the template window, and the message line.

The system offers the following facilities:

- setting and adjusting the focus (Section 4.1.1);
- copying and replacing the focused text (Section 4.1.1);
- replacing a focused placeholder by a syntactically correct template (Section 4.1.2);
- inserting and modifying the focused text (Section 4.1.3);
- undoing edit actions (Section 4.1.4);
- reading and writing files (Section 4.1.5);
- adjusting the unparsing of syntactical constructs (Section 4.3.4).

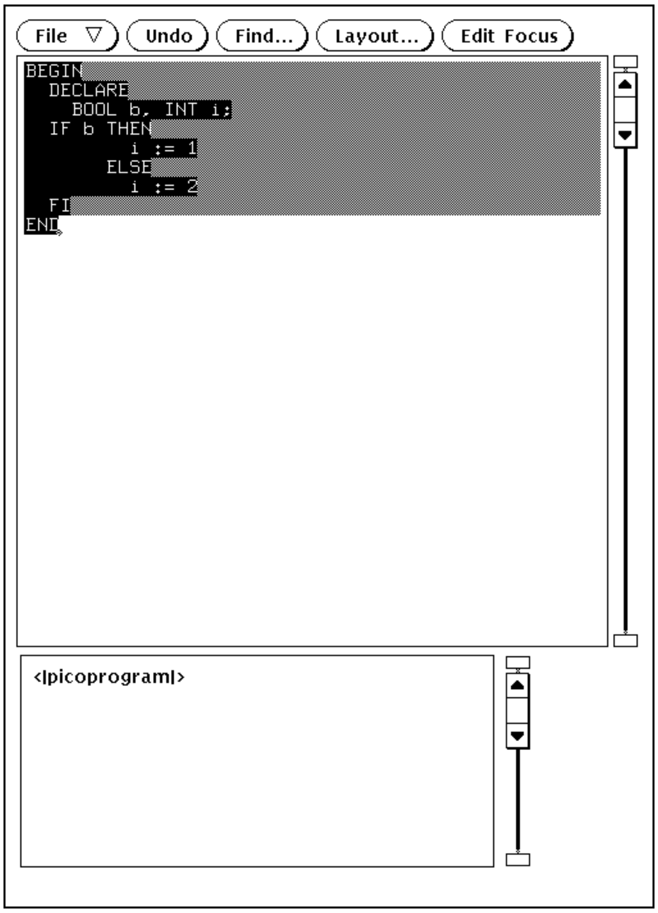


Figure 4.1: Main window.

4.1.1 Focus manipulation

Initially, the 3 subwindows of the main window are empty. An edit session can be started in two different ways, by reading a file or by inserting text via the text edit window. In both cases the text will be parsed, the corresponding tree-graph will be built and evaluated. As a result the focus window will then contain the unparsing of this tree-graph. Initially the complete program will be in the focus. The non-terminal of the syntactical construct in the focus is always given as the first template in the template window.

The *focus* is a syntactic piece of the program selected by the user, which is highlighted in the focus window. At least a part of the focus is always visible, except when the focus window is empty or the focus has been scrolled, by the user, outside the range of the focus window. The highlighted piece of program will be called the *extent* of the focus. So, the focus is a (sub)tree and the extent is its yield (or frontier).

Each character in the focus window is implicitly linked to a node in the tree-graph. The user can move the focus through the tree-graph. Pointing at characters outside the extent of the focus moves the focus up the tree-graph, to the common father of the old focus and the character pointed at. Pointing at characters inside the extent of the focus moves the focus down in the tree-graph. The smallest syntactical construct containing the character pointed at becomes the new focus. It is impossible to focus on layout characters, to select a subpart of a terminal symbol or to focus on a subpart of a semi-terminal.

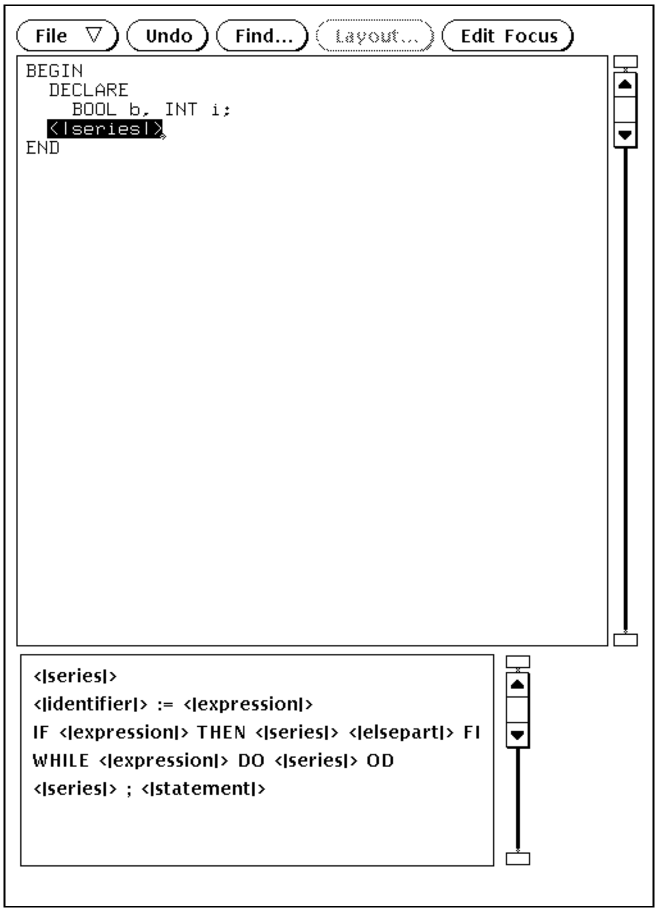


Figure 4.2: Main window; focus on a placeholder.

It is not possible to insert or delete a character in the focus window, the text editing facilities are restricted to the text edit window and will be discussed in Section 4.1.3. The focus window offers a saving-facility, viz. the possibility of saving the extent of the focus on a clipboard, and of retrieving the contents of the clipboard later. This can be done by a replacing-facility, viz. the possibility of replacing the extent of a new focus by the contents of the clipboard, or by a switching-facility, viz. the possibility to replace the extent of a new focus by the contents of the clipboard and saving the replaced text on the clipboard. Only the textual representation of the subtree will be stored on the clipboard, not the subtree itself. After replacing the extent of the new focus by the contents of the clipboard, the parser is called to analyze the resulting program.

4.1.2 Template facility

Templates are always available in our generated hybrid editor. Each time a new focus is selected the contents of the template window are refreshed. For most subtrees selected the template window will contain only one element, the placeholder of the syntactical construct which is focused. In this way the user always knows the non-terminal of the focused syntactical construct. If the focus is a placeholder the template window also contains all templates which are syntactically correct replacements of this placeholder (Figure 4.2). If the user is focused on the placeholder of a semi-terminal the template window will be empty except, of course, for the first element.

The user can select one of the templates to replace the extent of the focus. The selection of one of the templates always results in a syntactically correct program, in a few cases, however, the program will not be semantically correct. During the construction of the list of templates the values of the affix graph nodes connected to the affix position slices of the placeholder node are *not* taken into consideration. This information could filter out the templates which would yield a semantically incorrect program, but has not been implemented in the prototype of PREGMATIC.

A template is selected by clicking on it in the template window. The extent of the focus is then replaced by the textual representation of the template and the program is reparsed.

If the user focuses on a placeholder of a semi-terminal the text edit window is automatically popped up since this is the only way of transforming a semi-terminal placeholder into a real semi-terminal.

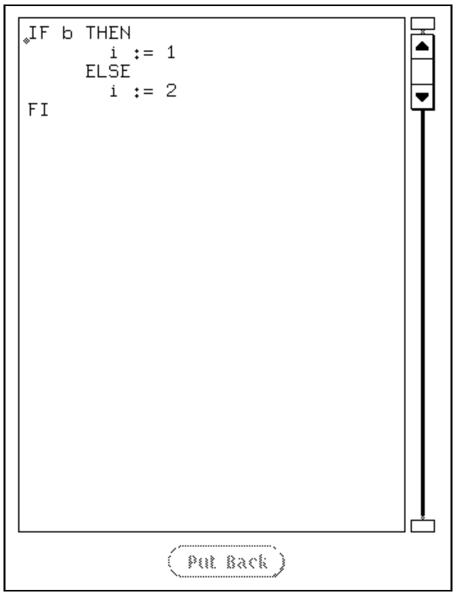


Figure 4.3: Text edit window.

4.1.3 Text editing

The text edit window can be explicitly invoked by clicking on the `Edit Focus` button (Figure 4.3) or implicitly by focusing on a semi-terminal placeholder. It will only contain the extent of the focus. Changing the focus results in an update of the contents of this window. This window offers the user a plain text editor with no knowledge of the syntactic structure of the text. It is possible to use the ordinary cut, copy, and paste facilities of the underlying text editor. The text edit session is finished by clicking on the `Put Back`-button, which is activated after the first alteration of the text in the text edit window. The extent of the focus will be replaced by the new contents of the text edit window and the resulting program will be reparsed. To ensure the maximum of incrementality a few extra facilities are included. These will be discussed in Chapter 5. If the user changes the focus after editing the text in the text edit window but before clicking on the `Put Back`-button, the edit actions are ignored.

It is not possible to use the contents of the clipboard of the focus window in a text edit session.

4.1.4 Undo

To ensure a flexible system the editor generated by PREGMATIC offer an undo-facility. The user can always undo the last replacement of the extent of the focus. The old extent of the focus is put back and the program is reparsed.

4.1.5 IO-facilities

The IO-facilities in the system are rather straightforward. It is only possible to read and write files. If the user reads a new file the complete tree-graph, if present, is replaced by the tree-graph built for the contents of the file read.

If the user uses the write facility the complete unparse of the tree-graph is written to the specified file. It is not possible to save subparts of the tree-graph.

A desirable, but not yet implemented, facility is to store the complete tree-graph rather than the unparse of the tree-graph. This would considerably speed up the processing of files.

4.2 Language-dependent environment issues

Until now the discussion of the PREGMATIC-EAG-formalism, the affix evaluation mechanism, and the user-interface have not specifically addressed a number of important issues related to the generation of programming environments. In this section we discuss:

- error handling,
- placeholders,
- templates.

4.2.1 Error handling mechanism

How can the error messages generated for erroneous input sentences be directly derived from the EAG-formalism? The technique we describe was initially developed to improve the error messages of the Programmar [BLM89].

Instead of finding as many errors as possible, we concentrate only on reporting in an informative way the first error encountered. The system works incrementally, so the amount of reparsing after corrections will be limited. We will here not only concentrate on the implementation, but also on the derivation of the contents of the error messages from the EAG specification.

The error handling mechanism is completely integrated into the system. The specification writer cannot influence it, he can influence the contents of the error messages only by renaming the non-terminals in his specification. The error handling is directed towards the user of the generated environment rather than the specification writer.

Since the parsers use affix-directed parsing we are able to treat type errors on an equal footing with syntax errors. In both cases our error handling mechanism may be expected to generate a decent message.

It is our intention to report

- how far in the input the parser advanced,
- whether a syntax or type error occurred, and
- why the parsing failed.

One of the problems we have to solve is that the parsers are based on backtracking. During the recognition of the sentence error messages will be generated at several positions, but only the messages generated for the most advanced position of the parser are relevant for the user. We have the *valid prefix* property, i.e. if the parser succeeds in recognizing a prefix this will be a prefix of a correct program. In some cases the source of the error lies at a less advanced position, for example in those situations where a begin marker of a syntactical construct is missing [LDHH78]. In these situations the error handling mechanism reports the error at the end of the syntactical construct rather than the lack of it at the beginning. This is, however, a failing common to most mechanisms.

We say that a predicate which is applied in the right hand side of a rule of which the left hand side is not a predicate is in a top-most position and call it a *top-most* predicate occurrence. Only these predicates will be used in the type error handling mechanism.

One part of the error message is the input position z. A second part is formed by the symbols $a_1, \ldots, a_n$ expected but not found by the parser and/or the names of the predicates $P_1, \ldots, P_p$ which all yielded `false` for the input position.

Note that due to the exhaustive search of the backtrack parser several symbols may be reported as possible candidates for further recognition. The same holds for the predicates.

The error position, the expected symbols and failing predicates already give a fair description of the error, but the message can be improved by mentioning the non-terminal of some syntactical construct surrounding the error position.

Consider the situation where the parser could not recognize the expected symbol a_j. All the non-terminals on the path from the root to a_j are surrounding constructs. The non-terminal closest to a_j on that path of which at least one member which is not a predicate has been recognized is what we call the *active syntactical construct* in which an error was detected. The non-terminal of the active syntactical construct will also be used in the generated error message. Consider the erroneous assignment statement

```
x = 1
```

where the =-symbol should have been :=-symbol. The generated error message indicates the =-symbol as erroneous and generates a message like

```
:=-symbol expected in assignation
```

This is an example of a typical syntax error message. Consider the erroneous declaration statements

```
DECLARE BOOL b, INT b;
```

The generated message indicates the second `b` as erroneous and generates a message like

```
enter declaration failed in declarations
```

The contents of an error message are based on the names of non-terminals and top-most predicate occurrences. This is therefore an additional factor for the specification writer to take into consideration. She does not, however, have to worry about error handling in her specification.

No tree-graph can be built for a substring containing an error. We do not want to force the user to correct his errors immediately. The erroneous string therefore is included as a special subtree in the tree-graph. This special subtree consists of two nodes, one top node containing the non-terminal of the expected syntactical construct and a leaf node containing the erroneous text.

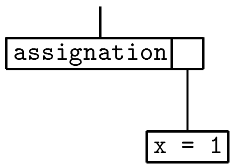


Figure 4.4: Error subtree.

The rest of the tree-graph considers the root node of the erroneous subtree as an ordinary tree node, therefore this root node should have the same affix positions as the tree node replaced. The values of the affix graph nodes connected to the lower sides of these affix position slices are $\perp$-values, as will be explained further in Section 5.2.

Implementation of syntax error handling

The error messages generated by the system consist of three parts:

- the error position;
- the expected symbol, or the top-most failing predicate occurrence;
- the active syntactical construct.

For syntax error messages we are only interested in the expected symbols and not in the top-most failing predicate occurrences.

The implementation of the first part of the error message is done by maintaining a provisional error position. The second part is implemented by calling a routine when recognition of a symbol fails. This routine compares the current error position with the provisional one. If the current position is less than the provisional one, the message is ignored. If it is equal the message is recorded. If the position is higher, previously recorded messages are removed and only the new message is kept.

The non-terminal of the active syntactical construct can be found on top of the tree node stack on which incomplete tree nodes are stored. The tree nodes of the non-terminals of which at least one member has been recognized are stored on this stack. If an error is detected, the non-terminal belonging to the node on top of this stack is reported as the active syntactical construct.

The number of messages can be reduced considerably by combining all expected symbols for the same syntactical construct in one message. Furthermore, (typed and untyped) placeholders are never reported as expected symbols. Consider the following erroneous PICO program, in Appendix C the eag for PICO can be found:

```
BEGIN
  DECLARE
    BOOL b, INT i;
  If b
  THEN i := 1
  FI
END
```

The parser is not able to recognize the `If`-symbol and will pinpoint its position as erroneous. The error message eventually generated is:

```
identifier, IF-symbol, or WHILE-symbol expected in program
```

Implementation of type error handling

The type error handling mechanism described in [BLM89] can also be used in the parsers of the generated environments.

A syntax error can be detected by the fact the parser fails to recognize some symbol and is not able to increase the input pointer. A type error is harder to determine, since such an error is raised by either:

- failure of a top-most predicate occurrence, which is always caused by the failure of *primitive* predicates,
- violation of a consistent substitution constraint, or
- failure of the recognition of an affix value by a defined affix non-terminal.

We will give possible error messages for each of these situations.

Predicates The implementation of the error handling for a failing predicate requires a method for recording the name of the top-most predicate occurrence, the non-terminal of the active syntactical construct in which the top-most predicate occurrence was called, and the non-terminal of the active syntactical construct in which the error was detected.

Predicates may be delayed during parsing and evaluated later. The non-terminal of the active syntactical construct in which the top-most predicate occurrence was called is needed for localizing the failing predicate, because the same predicate may be called several times. This name will be stored in the tree node of the top-most predicate occurrence. Evaluation of a delayed predicate causes the 'activation' of both its name and the stored non-terminal of the active syntactical construct. The non-terminal of the active syntactical construct in which the delayed predicate is evaluated is obtained in the same way as for the syntax errors. These three names are used to generate the error message.

Consistent substitution As soon as a value v is propagated to an affix graph node containing a value v' which differs from value v, the consistent substitution constraint is violated. Before the affix value propagation mechanism starts to backtrack, an error routine is activated, which generates a message

```
inconsistent affix values
```

together with the non-terminal of the active syntactical construct. If this inconsistency is detected during the evaluation of a predicate the message is suppressed and overruled by the message reporting the failure of the top-most predicate.

Defined affix non-terminals Each affix value v propagated to an affix graph node which represents a defined affix non-terminal A, is checked to determine whether it can be recognized by this affix non-terminal. If the non-terminal is not able to recognize the value, an error routine is called which generates a message

`affix non-terminal` A `failed to recognize an affix value`

together with the non-terminal of the active syntactical construct. Again, if this is detected during the evaluation of a predicate the message is ignored by the system.

4.2.2 Derivation of placeholders

In Section 1.4 we introduced typed and untyped placeholders, in this section we will discuss how almost each rule in an eag is extended with extra alternatives to offer the user of the generated environment the possibility of working with these placeholders.

Typed placeholders

The parsers in the editors generated by the Synthesizer Generator [RT89a] are not able to recognize typed placeholders. However a simple extension of the specification makes recognition of these placeholders possible. As an example we modify the SSL specification of a simple desk-calculator given in Appendix A of [RT89b]. We will give the new 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 this typed placeholder in text edit mode. In the Synthesizer Generator [RT89a] however the specification writer has to do this adaptation by hand.

Our system implicitly transforms it instead of letting the specification writer rewrite the specification. Each rule is extended with an extra alternative which recognizes the typed placeholder.

$$\mathtt{A} : \ldots\ . \Rightarrow \begin{cases} \mathtt{A{:}{<}|A|{>}}. \\ \mathtt{A} : \ldots\ . \end{cases}$$

The member in the right hand side of the new alternative can be considered as a terminal symbol.

Although a program text containing placeholders is incomplete, we want to analyze as much of the static semantics as possible. In order to be able to perform affix evaluation, it is necessary to assign values to the affix positions in the display of the non-terminal in the typed placeholder alternative of the rules. The solution chosen in the Synthesizer Generator is again cumbersome for the specification writer because he has to give a value to the attributes of the placeholders in the SSL specification. In our system this is done implicitly. The value has to represent all possible affix values, essentially Σ_A^*. Initially the affix value $\Box$ is assigned to all affix positions of the placeholder alternative. The affix value $\Box$ represent any legal affix value.

The internal representation of the rule `identifierlist` will be:

```
identifierlist (□):
  <|identifierlist|>.
identifierlist (decls):
  identifier (name),
    enter declaration (name, nil, decls).
identifierlist (decls):
  identifier (name),
    enter declaration (name, rest decls, decls),
      ",", layout,
        identifierlist (rest decls).
```

Predicates are excluded from extension with the typed placeholder alternative, nor can a typed placeholder replace a terminal symbol.

Untyped placeholders

The introduction of typed placeholders increases the flexibility of the editor, but in order to use them the user must know the exact names of all placeholders. This would be impossible for a language such as Algol68 if the user has to reproduce the non-terminal names used in [WMP+76].

We therefore allow the user to use the placeholder without the name of the corresponding non-terminal. Such an *untyped* placeholder represents almost all non-terminals in the language.

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

$$\texttt{A} : \ldots\ . \Rightarrow \begin{cases} \texttt{A:<|>.} \\ \texttt{A:<|A|>.} \\ \texttt{A} : \ldots\ . \end{cases}$$

This extension in general makes the grammar ambiguous. It is therefore not possible to adapt the SSL specification, as the generated parsers do not allow ambiguous context-free grammars.

The problem of ambiguity can be tackled in two ways. One may either use a more powerful parsing method or try to formulate criteria for extending the rules with the untyped placeholder alternative which ensure non-ambiguous grammars. Unfortunately, formulating a consistent set of rules for extending the rules turned out to be impossible. One might think of a combination of both techniques, but for the implementation of the prototype we have chosen otherwise.

Untyped placeholders never replace terminals or predicates. The affix positions associated with untyped placeholders are treated in the same way as those associated with the typed ones. So, the ultimate internal representation for the rule `identifierlist` is:

```
identifierlist (□):
  <|>.
identifierlist (□):
```

```
  <|identifierlist|>.
identifierlist (decls):
  identifier (name),
    enter declaration (name, nil, decls).
identifierlist (decls):
  identifier (name),
    enter declaration (name, rest decls, decls),
      ",", layout,
        identifierlist (rest decls).
```

4.2.3 Templates

Templates are not explicitly specified but are derived from the underlying context-free grammar. We will only derive templates for non-terminals which are not semi-terminals. Layout non-terminals and predicates are excluded from the template mechanism.

The set of templates of a non-terminal A is denoted by $\mathcal{T}(A)$. Consider, for example the rule for `assignation`.

```
assignation:
  identifier,
    check application,
      ":=", layout,
        expression.
```

The occurrences of the non-terminal `layout` and of the predicate `check application` are omitted. The resulting template is thus:

$\mathcal{T}$`(assignation) = {<|identifier|> ":=" <|expression|>}`

One way to derive the set of templates for each non-terminal is to collect the templates corresponding to the right hand sides of the individual alternatives of the non-terminal. The right hand sides of semi-terminals are *not* transformed into templates. The templates for the rules for `expression`, `term` and `factor` would be:

$\mathcal{T}$`(expression) = {<|term|>;`
`                 <|expression|> "+" <|term|>}`

$\mathcal{T}$`(term) = {<|factor|>;`
`           <|term|> "*" <|factor|>}`

$\mathcal{T}$`(factor) = {<|identifier|>;`
`             <|number|>}`

This is not an optimal solution for the user. It is possible that the user has a specific construct in mind and then has to perform a lot of unnecessary transformation steps. If the user wants to transform the placeholder <|expression|> into the typed placeholder <|identifier|> he needs several steps.

This tedious way of developing programs is prevented by replacing the templates generated for alternatives which would consist of one non-terminal (*chain rules*) by the set of templates of this non-terminal. This is done recursively.

The sets of templates for the non-terminals `expression`, `term` and `factor` using the strategy described above are now:

```
𝒯(expression) = {<|identifier|>;
                 <|number|>;
                 <|term|> "*" <|factor|>;
                 <|expression|> "+" <|term|>}

𝒯(term) = {<|identifier|>;
           <|number|>;
           <|term|> "*" <|factor|>}

𝒯(factor) = {<|identifier|>;
             <|number|>}
```

4.3 Unparsing

In this section we concentrate on the unparsing mechanism of our syntax-directed editors. The mechanism is based on the underlying context-free grammar of an eag, but it will work for any context-free grammar.

The generator extracts all unparsing information from the context-free grammar itself (without any additional unparsing rules) and generates a list of tuples which is used by a language-independent unparser. We do allow the user of the editor to change the contents of this list (Section 4.3.4).

The layout of a program is a very personal matter and no general pretty print strategy will satisfy all users of the editor. Editors for different languages produce similar layout for programs. However, different languages may ask for completely different unparsing rules. Our mechanism will work satisfactorily for languages of the Algol-family, but for functional languages or syntax-based languages like CDL3 [KB91] the unparsing may be awkward. Anyway the specification writer need not bother himself with unparsing when prototyping a language.

The unparsing mechanism described in [BS89] makes it possible for the user of the editor to influence the unparsing process directly. This is done by allowing the user to adjust the value of a number of variables, such as LineWidth, Indentation, etc. This unparsing mechanism is strongly connected to the languages Pascal and Modula-2. It has no facilities for adapting the unparsing of an arbitrary syntactical construct and it cannot be used in generated programming environments. The unparsing mechanism is based on Oppen's algorithm [Opp80].

The system we developed is also strongly influenced by the algorithm formulated by Oppen. His mechanism allows the specification writer to insert output indications in the specification, which the algorithm uses when traversing the abstract syntax tree to generate a pretty print. In articles of later date this approach has been investigated further and described [RW81, Mat83, Rub83, Lea84, BS84, Woo86], but there has been no attempt to formulate criteria for language-independent unparsing — not even in [Jok89].

An unparser must produce a pleasantly readable layout of a program text. The readability of a program text is increased by spreading large syntactical constructs over several lines in a structured and consistent way. Small syntactical constructs which fit on one line must not be split. The unparsing of a syntactical construct is primarily influenced by the number of characters left on a line.

Our discussion is now split into three parts: first we give the algorithm which takes an unparsing specification and a tree-graph and produces the unparsing of that tree-graph. Secondly, we describe how an unparsing specification is generated starting from a context-free grammar. Thirdly, we describe the possibilities which the system offers the user for changing the unparsing of the syntactical constructs.

4.3.1 The algorithm of Oppen

Oppen's algorithm [Opp80] receives a list of lexical tokens together with special characters to direct the unparsing. There are two categories of special characters:

- *bracket characters* to delimit a syntactical construct, such as an assignment or a series;
- *blank characters* to mark a possible line break and/or the number of blanks to be printed between lexical symbols.

The bracket characters consist of an open bracket, written as ⦃, to denote the start of a syntactical construct and a close bracket, written as ⦄, to denote its end. The blank character, written as $\square$, denotes a possible break points. For example,

```
⦃IF x = y □ THEN x := 0 □ ELSE x := 1 □ FI⦄
```

will be unparsed as:

```
IF x = y THEN x := 0 ELSE x := 1 FI
```

provided this syntactical construct fits on the remaining space of a line. If, however, it does not fit, the syntactical construct will be spread over several lines. The blank characters may be either consistent, $\square_{\mathtt{C}}$, or flexible, $\square_{\mathtt{F}}$. Consistent blanks have the property that for one construct either all of them are replaced by a newline plus an indentation, or some of them are replaced. The syntactical construct above, with all the $\square$-symbols replaced by $\square_{\mathtt{C}}$-symbols, would then be unparsed as:

```
IF x = y
  THEN x := 0
  ELSE x := 1
  FI
```

The flexible blank type indicates that the blank characters need not be replaced by a newline. If the rest of the structure after a blank character fits on the remaining space of a line it will be unparsed on this line. The syntactical construct above, with all $\Box$-symbols replaced by $\Box_{\mathtt{F}}$-symbols, can then be unparsed as:

```
IF x = y
  THEN x := 0
  ELSE x := 1 FI
```

or as:

```
IF x = y
  THEN x := 0 ELSE x := 1 FI
```

However, Oppen is quite vague about the way to specify and obtain a layout like:

```
IF x = (y + 1) THEN x := 0
               ELSE x := 1
               FI
```

where the `ELSE` and the `FI` are always related to the beginning of the `THEN`-symbol.

One of the drawbacks of this mechanism is the tedious way of producing the unparsing. Firstly, the tree-graph is traversed to produce the list of lexical tokens which has to be buffered. Secondly, the length of each syntactical construct is calculated and then the unparsing is produced using an extra stack. This makes the algorithm less suited for implementation in a programming environment.

4.3.2 The unparsing algorithm

The unparsing algorithm used in PREGMATIC uses also two passes. In the first pass the length of each syntactical construct is calculated and in the second pass the layout of the syntactical constructs is determined and the unparsing of the tree-graph is written into a buffer, which is then used to refresh the screen.

The unparsing algorithm is based on a column and line administration of an imaginary screen. The screen width is known and will not change during the unparsing of the tree-graph.

A relative column- and line-offset is associated with each member in a right hand side: the column-offset and line-offset are added to the current screen position and the member will be printed at the new position. A member may be 'position related' to a preceding member. In that case, the current position is first reset to the column value of the related member and the line value is not modified.

If '$i : (M,\ j,\ c,\ l)$' is a tuple in the unparsing specification list then i represents the i^{th} member in an alternative, j represents the j^{th} member to which the i^{th} member is related. If $i = j$ then the i^{th} member is not related to any other member. The value j must always be smaller or equal to i. The symbol M is either a terminal symbol, or a typed placeholder, if the i^{th} member is a non-terminal. The symbols c and l represent the relative column- and line-offset respectively. The column-offset c and line-offset l must both be greater than or equal to zero.

Suppose we have the following rule:

```
ifstatement:
  "IF",
    expression,
      "THEN",
        series,
          "ELSE",
            series,
              "FI".
```

the unparsing specification list entry will be:

```
0: (IF,             0,  0,  0)
1: (<|expression|>, 1,  1,  0)
2: (THEN,           0,  2,  1)
3: (<|series|>,     3,  1,  0)
4: (ELSE,           0,  2,  1)
5: (<|series|>,     5,  1,  0)
6: (FI,             0,  0,  1)
```

the unparsing will be:

```
IF <|expression|>
  THEN <|series|>
  ELSE <|series|>
FI
```

There are two entries in the unparsing specification list for each alternative of a rule: one for horizontal unparsing and one for vertical unparsing. These two may be the same.

The tree traversal algorithm, which calculates lengths, records the sizes for the *horizontal* unparsing of each syntactical construct in the corresponding tree node. This routine traverses the tree-graph and in each node considers the corresponding horizontal unparsing entry in the unparsing specification list. The width and height of the horizontal unparsing is calculated using the information of this entry.

The tree traversal algorithm for unparsing the tree-graph also traverses it and unparses each node, using the stored length information. If the length is greater than the space left on the current line then the vertical unparsing entry is chosen instead of the horizontal one to produce the unparsing of this node.

In left- and right-recursive rules with different vertical and horizontal unparsing rules this strategy may lead to an irregular unparsing. Consider the rule:

```
series:
  statement,
    ";",
      series.
series:
  statement.
```

The unparsing of a list of statements may then be:

```
<|statement|>;
<|statement|>; <|statement|>
```

The space left on the current line was too small to unparse the 3 statements horizontally, but it was too large for a complete vertical unparsing. Forcing a consistent vertical unparsing can be done in two ways. Either by having the horizontal unparsing equal to the vertical one which will result in a vertical unparsing in *all* cases or by indicating that the unparsing of the `series` in a vertical unparsing should always be unparsed vertically. This is done by inserting a 'force-vertical character', `#`-character, in front of the `<|series|>` in the vertical unparsing specification list entry: `(#<|series|>,...)`.

In Section 4.1.1 we remarked that each character in the focus window is directly linked to the tree-graph. It is therefore necessary for the unparsing algorithm to store a link to the corresponding tree node for each symbol written in the screen buffer. These links are stored in a so-called focus buffer, which is used, amongst other things, by the focus mechanism. The screen buffer is used to refresh the contents of the windows.

4.3.3 Generation of unparser

The generation of the unparser corresponds to generating the entries in the unparsing specification list, both horizontal and vertical. The unparser generator is in fact a simple transducer. Given an alternative of a rule the entries in the unparsing specification list for the corresponding node in the tree-graph are generated. The alternative:

$non\text{-}terminal_0$:
  $member_1$, ..., $member_n$.

has the following horizontal unparsing specification list entry:

0:   (<|$member_1$|>, 0, 0, 0)
1:   (<|$member_2$|>, 1, 1, 0)
     ⋮
n-1:(<|$member_n$|>, n-1, 1, 0)

and the following vertical unparsing specification list entry:

```
0:   (<|member_1|>, 0, 0, 0)
1:   (<|member_2|>, 1, 0, 1)
        ⋮
n-1:(<|member_n|>, n-1, 0, 1)
```

This transduction scheme is far too simple, but extra information is needed to generate better unparsing rules.

The generator first assigns a type to each alternative. We only consider the underlying context-free grammar of the eag, i.e. with all predicate calls and all non-terminals which describe layout filtered out. There are three different types of alternatives:

1. the *bracket type* for alternatives with a bracket structure, i.e. each alternative consists of at least two members and the first and/or the last member is a terminal;

2. the *recursive type* for alternatives which are left- or right-recursive, i.e. each alternative consists of at least two members and the first and/or the last member equals the non-terminal in the left hand side;

3. the *untyped type* for all other alternatives.

The alternatives of the first type have the form:

- *non-terminal*$_0$:
  "*terminal*$_1$", ..., "*terminal*$_n$".

- *non-terminal*$_0$:
  "*terminal*$_1$", ..., *non-terminal*$_n$.

- *non-terminal*$_0$:
  non-terminal$_1$, ..., "*terminal*$_n$".

It may be strange to consider the last two types of alternatives as bracket alternatives, but they can be considered as alternatives of the form:

- *non-terminal*$_0$:
  "*terminal*$_1$", ..., *non-terminal*$_n$, "".

- *non-terminal*$_0$:
  "", *non-terminal*$_1$, ..., "*terminal*$_n$".

The alternatives of the recursive type have the form:

- *non-terminal*$_0$:
  non-terminal$_1$, ..., *non-terminal*$_0$.

- *non-terminal*$_0$:
  non-terminal$_0$, ..., *non-terminal*$_n$.

The third type is for alternatives of the form:

- *non-terminal*$_0$:
  non-terminal$_1$, ..., *non-terminal*$_n$.

  where

  $$non\text{-}terminal_0 \neq non\text{-}terminal_1$$
  $$non\text{-}terminal_0 \neq non\text{-}terminal_n$$

In addition to this type information it is also necessary to know which non-terminals have an empty alternative, and the length of each terminal. With this information we state the following rules.

Rule 1:

The column and line value of the second tuple in the unparsing specification list entry of an alternative which begins with a terminal symbol consisting of one character only will not be increased. The column and line value is not increased for tuples of terminals of length one which are not the first member in an alternative. The horizontal and vertical tuple entries for an alternative like:

```
subscription:
  "[", expression, "]".
```

are identical and look like:

```
0:  ([,              0, 0, 0)
1:  (<|expression|>, 1, 0, 0)
2:  (],              2, 0, 0)
```

This rule will always be applied, independently of the alternative type. The tuples for these terminals are not related to any tuple in the unparsing specification list entry.

Rule 2:

The rule for vertical tuples for alternatives of bracket type consists of two parts:

1. The column and line values of tuples for terminals of length greater than one preceded by a non-terminal are 0 and 1 respectively.

2. The column and line values of tuples for non-terminals preceded by a terminals of length greater than one are 2 and 1 respectively.

All tuples generated by one of these two rules are related to the first tuple of the unparsing specification list entry.

The rule for horizontal tuple entries for these alternatives is straightforward. The column value of tuples for terminals of length greater than one, preceded by a non-terminal as well as for non-terminals preceded by a terminal of length greater than one, is 1. The tuples are *not* related to any other tuple in the unparsing specification list entry. An alternative like:

```
whilestatement:
  "WHILE", expression, "DO", series, "OD".
```

has the following vertical unparsing specification list entry:

```
0:  (WHILE,          0, 0, 0)
1:  (<|expression|>, 0, 2, 1)
2:  (DO,             0, 0, 1)
3:  (<|series|>,     0, 2, 1)
4:  (OD,             0, 0, 1)
```

Rule 3:

The rule for the vertical tuples for alternatives of recursive type is: the column and line values of all tuples, except for the tuples which satisfy Rule 1, are 0 and 1 respectively. These tuples are *all* related to the first tuple in the entry. For the horizontal tuple entry these values are 1 and 0 respectively. These tuples are *not* related to any other tuple.

A recursive rule like:

```
series:
  series, ";", statement.
```

has the following unparsing specification list entry:

```
0:  (<|series|>,    0, 0, 0)
1:  (;,             1, 0, 0)
2:  (<|statement|>, 0, 0, 1)
```

This unparsing rule could lead to the following layout, as we have seen in Section 4.3.2:

```
<|statement|>; <|statement|>;
<|statement|>
```

This can be prevented by inserting the #-character in the tuples of recursive non-terminals in the vertical unparsing specification list entry. In the prototype the user of the editor has to insert this #-character.

Rule 4:

If, in a rule of bracket type, two non-terminals are not separated by a terminal, and one of the two or both may produce empty, the generator will only increase the line value of the tuple in the vertical unparsing specification list entry belonging to the second non-terminal and relate it to the first tuple in the unparsing specification list entry. In the horizontal unparsing specification list entry neither the column nor the line value of this tuple is increased and the tuple is not related to any other tuple. The vertical unparsing specification list entry for an alternative like:

```
ifstatement:
  "IF", expression, "THEN", series, elsepart, "FI".
```

where:

```
elsepart:
  "ELSE", series.
elsepart:
  .
```

will be:

```
0:  (IF,             0, 0, 0)
1:  (<|expression|>, 0, 2, 1)
2:  (THEN,           0, 0, 1)
3:  (<|series|>,     0, 2, 1)
4:  (<|elsepart|>,   0, 0, 1)
5:  (FI,             0, 0, 1)
```

Rule 5:

The horizontal and vertical unparsing specification list entries for the alternatives of semi-terminals are very simple. *No* layout is inserted between the members of these alternatives, so the column and line values in these tuples are 0 and the tuples are not related to each other.

Heuristic rules

The rules above have a very strong heuristic nature. Other rules could of course be formulated. It is also possible to consider more characteristics of the grammar, such as the level at which a rule is applied.

4.3.4 Adaptation of unparsing rules

Finally, we want to discuss how the user can adapt the unparsing rules. Clicking on the `layout`-button will pop up the layout modification window, Figure 4.5.

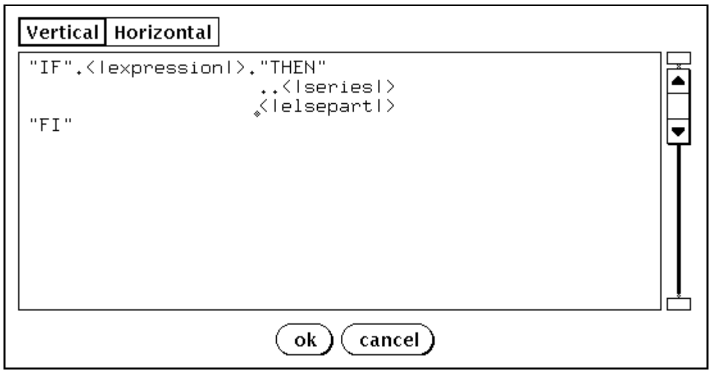


Figure 4.5: Layout modification window.

It makes it possible for the user to adapt the unparsing rule of the focused syntactical construct. If the syntactical construct cannot have unparsing rules, if for example it is a semi-terminal or a placeholder, the `layout`-button is not activated. The window contains the horizontal or vertical unparsing of the syntactical construct with each subtree replaced by its corresponding placeholder. It strongly resembles a template.

Only a few alterations are allowed. The unparsing of a syntactical construct can be adapted by inserting or deleting space characters, newline characters, and the `#`-character. It is not possible to modify the terminals and placeholders visible in the window. The `#`-character may only be inserted immediately in front of a placeholder. There are two space-symbols: the “`.`”-symbol, representing the space symbols in the unparsing of the corresponding node of the tree-graph, and the “ ”-symbol, used to shift terminal symbols and placeholders into the appropriate column. Any terminal or placeholder which is preceded by one or more “`.`”-symbols is considered to start in the column containing the first “`.`”-symbol. Consider Figure 4.5, where the `FI`-symbol is related to the `IF`-symbol, the placeholder `<|elsepart|>` is related to the `THEN`-symbol as is the placeholder `<|series|>` though it will always be preceded by two blanks. The resulting unparsing can be found in Figure 4.1.

If the user clicks on the `ok`-button the adaptation of the vertical or horizontal unparsing rules of a syntactical construct becomes immediately visible. If he clicks on the `cancel`-button the adaptation to the unparsing rule are ignored. The `horizontal` and `vertical` buttons are used to switch between both orientations. Switching has the same effect as clicking on the `cancel`-button. The adaptations made in the unparsing rules are permanent, quitting the editor will not restore the original unparsing rules.

Chapter 5

Incrementality

In Chapter 3 we assumed that the complete program text is reparsed after each edit action and a complete new tree-graph is built and evaluated. We show the sequence of steps to obtain a consistent tree-graph again.

1. Perform the edit action on the extent of the focused subtree-graph. This text can be changed by:

   - inserting new text before,
   - appending new text after, or
   - changing the text within,

   the focused syntactical construct. Incremental behaviour of the system can be obtained if only the smallest affected syntactical construct is reparsed.

2. Prune the 'modified' subtree-graph. It is necessary to remove part of the tree-graph in order to process the alterations in the text. This will be replaced by a new subtree-graph, obtained by reparsing the new text. The subtree is removed by simply cutting the tree link between the tree node F and the top node N of the subtree to be removed. The subgraph is removed by cutting all links between the upper side of the affix position slices of node N and the connected affix graph nodes.

   It is also necessary to propagate the $\perp$-value via the removed links of type 'first-in'. The $\perp$-propagation mechanism is described in Section 5.2.2. All affix graph nodes which depend on the removed subtree-graph, are marked by this process.

3. Reparse the modified extent of the pruned subtree-graph, starting at the non-terminal of the root of the pruned subtree-graph.

4. If the text is syntactically correct and no local context conditions are violated, the parser returns a subtree-graph. Syntactically the subtree-graph must fit in the same place in the tree-graph as the old subtree-graph. Otherwise a different (higher) tree node in the tree-graph would have been selected to start the reparsing. It is, however,

possible that the grafting causes a violation of some context condition; this will be discussed in Section 5.6.3.

The subtree is grafted by establishing one link only. The grafting of the affix subgraph consists of establishing several links between some graph nodes and the upper side of the affix position slices of the top node of the constructed subtree-graph. After establishing these links the affix value propagation mechanism is invoked to make the tree-graph consistent again. During this process only nodes with $\perp$-values are visited.

A type error may also be detected during the affix evaluation started after the grafting of the subtree-graph. The affix evaluation process will then start to backtrack.

In this chapter we will look at incremental reparsing, placeholder recognition, and incremental unparsing. We will also look at incremental affix evaluation and give a number of improvements to increase efficiency.

5.1 Reparsing

Altered text is reparsed by a routine which has two parameters:

- the text that is to be parsed, and
- the non-terminal to which the text should be reduced.

The user of a generated syntax-directed editor will always edit within the extent of the focus. An edit action can be seen as replacing the current extent of the focus by a new text. Although the rest of the tree-graph is not affected, starting the reparsing with the focused node and the new text may not guarantee a successful recognition. Suppose the user has focused on the expression `1` in the following program:

```
BEGIN
  DECLARE
    BOOL b, INT i;
  IF b
  THEN i := 1
  FI
END
```

The user now changes the extent of the focus into:

```
1 ELSE i := 2
```

Instead of reparsing only the altered expression, the reparsing process should start in the node the extent of which is the complete conditional, otherwise recognition will fail. Although in most cases the focused node will cover the alterations, in a few cases the reparsing will have to start in an ancestor node of the focus.

There is always a minimal node which covers the complete syntactical effect of the edit action. In [Bra90] this minimal node is found by some oracle, in reality the system can only determine this node by trial and error. One strategy to find this node is to try all nodes on the path from the focus to the root of the tree-graph.

Of course we cannot prevent the system from doing unnecessary work during the search for this minimal node. If the text to be parsed contains an error, a lot of unnecessary work may be involved. We will formulate heuristic rules to reduce the number of nodes to be tried by the system. All edit actions can be reduced to the following two cases:

1. A placeholder is replaced by a template: this operation preserves syntactic correctness.

2. Either a syntactical construct is replaced by another syntactical construct (by applying a replacing- or switching-facility) or its extent is textually modified; with both operations the syntactical correctness and structure are no longer guaranteed.

In the first case, if the reparsing of a template does not succeed it is not necessary to try another node, because the error is a type error.

In the second case, the edit actions may cause a complete restructuring of the tree-graph. Of course we can use the straightforward method of trying each node until a successful parse is found. But it is probably to be more efficient to try only a few nodes and then, if still not successful, to reparse the complete program. In our system the reparsing is first tried in the focused node. If this is not successful the *strictly enclosing syntactical construct*[1] is tried, and if this is still not successful the complete program is reparsed. Consequently, the complete program is reparsed for each syntax error.

There is a range of different techniques to determine where the reparsing process should start. We have selected just one, which works satisfactorily for our system. In the ASF+SDF meta-environment [Kli91], for example, only the extent of the focus is reparsed. If the parser finds an error, the user of the system has to move the focus explicitly.

5.2 Type checking

The affix evaluation mechanism described in Chapter 3 was not incremental. Before we give the details of the incremental affix evaluation mechanism, we describe the internal representation of affix values.

5.2.1 Representation of affix values

In Chapter 3 we discussed the affix value propagation mechanism without knowing what kind of values were propagated and what kind of operations were performed on these values in the affix position slices. In this section we give a description of the affix value types and their internal representation. We also describe the operations on them and pay attention

[1]A node n is a strictly enclosing syntactical construct of a descendant node n' if the extent of n is $\alpha\beta\gamma$ where β is the extent of n' and $(\alpha \neq \varepsilon \vee \gamma \neq \varepsilon) \wedge n \neq n'$.

to their effects on the special □-value and ⊥-value. The following operations are performed in the affix position slices:

- *tuple operation*,
- *concatenation operation*, and
- *no operation.*

Tuple operation If an affix expression contains a tuple operator then the affix values of affix graph nodes connected to an affix position slice (Figure 5.1) are either

- combined into a n-tuple affix value, or
- the n-tuple affix value is deterministically split into the n values.

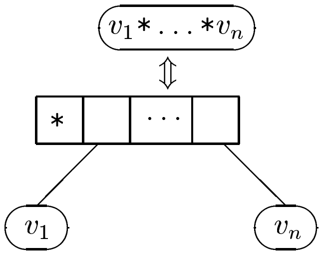


Figure 5.1: Tuple operation.

Concatenation operation If an affix expression contains a concatenation operator then the affix values of affix graph nodes connected to an affix position slice (Figure 5.2) are either

- added or concatenated to an affix value of type *NUMERAL* or *STRING*
- the affix value is non-deterministically split into n values.

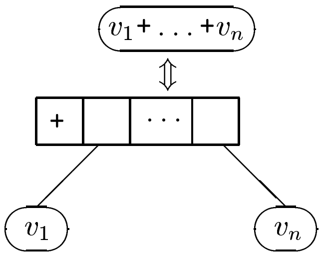


Figure 5.2: Concatenation operation.

No operation The affix value in Figure 5.3 is merely propagated, without any operation being performed on it.

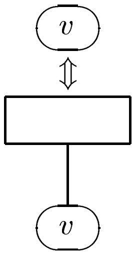


Figure 5.3: No operation.

For the affix value propagation algorithm described it was sufficient to know that each affix graph node contained either a defined value or no value at all. In Section 5.4.2 we will use the $\square$-value for the initialization of affix graph nodes connected to the affix position slices of placeholders. The notion of 'no value yet' is formalized in the $\perp$-value. An affix graph node with the $\perp$-value has not yet been visited by the evaluator. The initial value of all affix graph nodes except affix graph nodes containing a terminal affix value is the $\perp$-value. This value is also used for the initialization of affix graph nodes connected to the root nodes of erroneous subtrees. These two new values make it necessary to reconsider affix values. We will now describe their structure and the operations on them more explicitly.

The internal representations of the affix values of type *NUMERAL* and *STRING* are simple and consist of a numeral (≥ 0) or zero or more characters respectively.

An affix value of type *TUPLE* $v_1 * \ldots * v_n$ consists of a list of references to the elements of the tuple, see Figure 5.4.

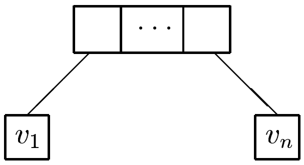


Figure 5.4: Tuple value representation.

where

$$v_i \in (NUMERAL \cup STRING \cup TUPLE \cup \{\square \cup \perp\}), 1 \leq i \leq n.$$

The $\square$-value and $\perp$-value represent the unspecified and the undefined value respectively. There is a difference with respect to the operations performed on them. The $\square$-value is more or less treated as an ordinary affix value, except for:

$$\mathtt{v_1+\ldots+\square+\ldots+v_n} = \square$$

The operation should be read from left to right. By allowing $\square$-values to be an element of a tuple, the affix evaluation mechanism is able to do as much type checking as possible.

The $\perp$-value is less defined than the $\square$-value and consider the following operations:

$$\mathtt{v_1+\ldots+\perp+\ldots+v_n} = \perp$$

$$\mathtt{v_1*\ldots*\perp*\ldots*v_n} = \perp$$

If at least one argument of a tuple operator has the $\perp$-value the resulting value is $\perp$-value.

The □-value is less defined than any affix value of type *NUMERAL*, *STRING*, or *TUPLE* (Figure 5.5). The consequence is that if the affix evaluation process propagates a value $\mathtt{v} \neq \Box$, and it reaches an affix graph node which contains the □-value it is replaced by `v`. The affix value propagation process then continues via the 'in' links in this node.

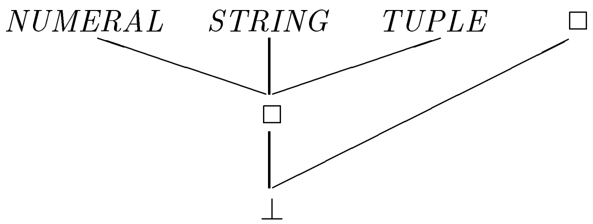


Figure 5.5: Pseudo-lattice of affix domain types.

5.2.2 Affix value propagation revisited

The incremental type checking mechanism is in fact equivalent to the evaluation mechanism described in Chapter 3, except for a few small changes in the affix value propagation algorithm. It is necessary to define a marking algorithm which traverses the tree-graph to mark the affected affix graph nodes.

Initially, almost all affix graph nodes have the ⊥-value. These nodes have to be visited by the affix value propagation algorithm in order to assign values to them.

After a subtree replacement, a number of affix graph nodes have to be re-evaluated. The modification to the tree-graph may affect not only the values of the affix graph nodes connected to the affix position slices of the changed tree node, but also the values of other affix graph nodes which (in)directly depend on these affected nodes. In [Rep84] an optimal re-evaluation algorithm is formulated, based on the dependency graph. The re-evaluation in Pregmatic does not use a dependency graph, the re-evaluation is completely dynamic. Two different strategies for re-evaluating affected affix graph nodes are presented here.

It is necessary to adapt the affix value propagation algorithm presented in Section 3.4.3 for both solutions. The incremental evaluation mechanism must know which link was responsible for the initialization of the affix graph node, otherwise too many nodes will be marked as affected, which results in a expensive re-evaluation process. This link is therefore marked as 'first-in' using the marker '`IN`', by the affix value propagation algorithm.

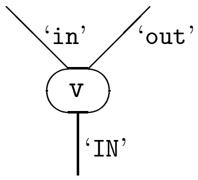


Figure 5.6: First-in link.

The ⊥-propagation algorithm

This is the less efficient of the two solutions because all affix graph nodes which may have been affected have to be visited by the marking- and affix value propagation algorithm, independently of the changes to their values. The marking algorithm is a propagation algorithm which propagates the $\perp$-value through the tree-graph. It starts propagating after pruning a subtree-graph.

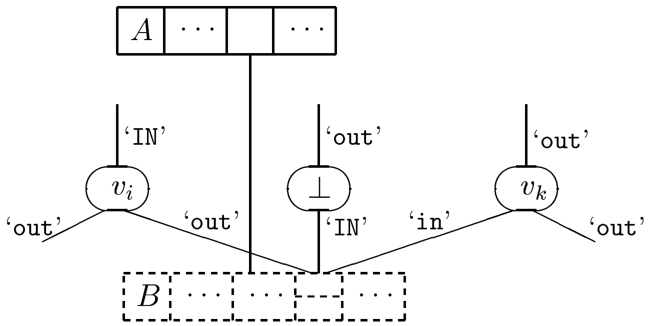


Figure 5.7: $\perp$-propagation.

In Figure 5.7 the dashed box represents the pruned tree node. The affix graph node containing the $\perp$-value was connected to an affix position slice of the pruned tree node via a 'first-in' link.

All affix graph nodes connected via 'first-in' links to the removed tree node are marked by the $\perp$-value.

Starting in an affix graph node A the $\perp$-value is propagated to the rest of the graph via the 'out' links of this node. The condition to continue the $\perp$-propagation process in an affix graph node is:

$$\exists j : 1 \leq j \leq \texttt{edge\_number\_a}(A) \wedge \texttt{type\_of}(\texttt{edge\_a}(A,j)) = \text{`}\texttt{out}\text{'}$$

The sill of the connected affix position slice side s is increased and the $\perp$-value is propagated to the other side s' of the affix position slice. The type of the link between A and side s is changed to 'undefined'.

The $\perp$-value is then propagated via all links connected to s' of type 'in' and 'first-in'. The sill of s' is adjusted. If no links of these types exist the $\perp$-propagation process stops. The condition for continuing the $\perp$-propagation process is therefore

$$\begin{array}{l} \exists j : 1 \leq j \leq \texttt{edge\_number\_p}(T,s') \\ \wedge \\ (\texttt{type\_of}(\texttt{edge\_p}(T,s',j)) = \text{`}\texttt{IN}\text{'} \vee \texttt{type\_of}(\texttt{edge\_p}(T,s',j)) = \text{`}\texttt{in}\text{'}) \end{array}$$

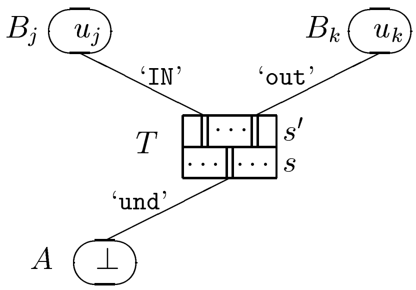


Figure 5.8: Affix position slice.

All affix graph nodes have at most one 'first-in' link. This link was responsible for the initialization of the node. If the $\perp$-value is propagated to an affix graph node via this 'first-in' link the value of the node becomes the $\perp$-value and the process is recursively continued. The propagation of the $\perp$-value via an 'in' link will not affect the value of the affix graph node and the process stops in this node. It is not necessary to continue the $\perp$-propagation in such a node because the value of this graph node depends on another part of the tree-graph. The type of the link is changed to 'undefined' in both situations. In Figure 5.8 the $\perp$-propagation process moves the $\perp$-value from affix graph node A to B_j, yielding Figure 5.9.

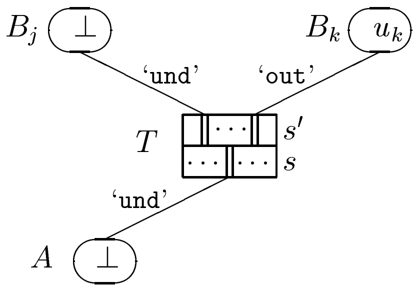


Figure 5.9: Resulting affix position slice.

If the type of the link had been 'in' the value of B_j in Figure 5.9 changed, only the type of the link to B_j would have been changed, not the value in B_j.

The two algorithms for propagating the $\perp$-value from tree node to graph node and from graph node to tree node are respectively.

```
⊥-propagate(slice,side)
{
 nr := edge_number_p(slice,side);
 for i := 1 to nr {
  node := affix_of(slice,side,i);
  edge := edge_p(slice,side,i);
  if (value_of(node) <> ⊥) {
   type := type_of(edge);
   if (type = 'IN') {
    type_of(edge) := 'u';
    value_of(node) := ⊥;
    ⊥-propagate'(node);
   }
   else if (type = 'in')
    type_of(edge) := 'u';
  }
 }
}

⊥-propagate'(node)
{
```

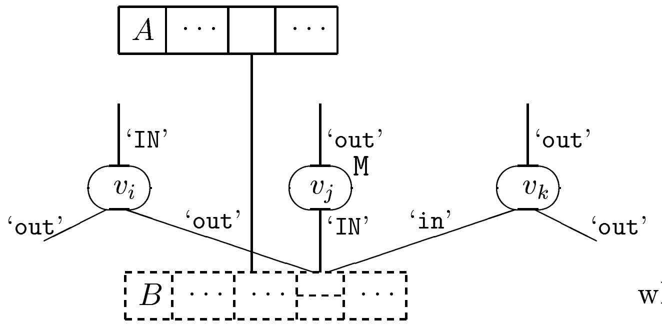
```
 nr := edge_number_a(node);
 for i := 1 to nr {
  edge := edge_a(node,i);
  if (type_of(edge) = 'out') {
   (slice,side,pos) := vertex_p(edge);
   type_of(edge) := 'u';
   ⊥-propagate(slice,(1-side));
  }
 }
}
```

Neither of the algorithms `⊥-propagate` and `⊥-propagate'` is based on backtracking. The reparsing of the altered text either succeeds, in which case the newly constructed subtree-graph is grafted onto the rest of the tree-graph and the tree-graph is again made consistent, or the parsing fails and the text is transformed into an erroneous subtree which is grafted onto the tree-graph. The affix graph nodes connected to the lower sides of the corresponding affix position slices of this erroneous subtree will all have the $\perp$-value and thus no extra propagation actions are needed. One of these two actions will always occur, so undoing the $\perp$-value propagation is not necessary.

The propagation of the $\perp$-value to a non-critical affix position slice of a predicate causes the propagation process to stop in this slice. If, however, the value is propagated to a critical affix position slice the predicate becomes delayed and the values of all affix graph nodes connected to the upper side of the non-critical affix positions become the $\perp$-value. These $\perp$-values are propagated through the rest of the affix graph.

The marking algorithm

A second solution is inspired by the optimal re-evaluation algorithm formulated by [Rep84]. It is a more efficient variant of the $\perp$-propagation algorithm. In the tree-graph from which the old subtree-graph is removed only the affix graph nodes which are connected to the removed tree node via a 'first-in' link are marked as affected, for example 'M' in Figure 5.10. These are not marked if they contain the $\perp$-value or $\Box$-value.



where $v_j \neq \Box$ and $v_j \neq \perp$.

Figure 5.10: $\perp$-propagation.

The new propagation algorithm will visit not only all nodes with the $\perp$-value but also the marked affix graph nodes. If it propagates a value `v` to a marked affix graph node, the old value of this node is compared with the propagated one. If these values are equal the propagation process stops at this node; if they are *not* equal, the old value is overwritten by the propagated value and the propagation process is continued via all 'out' links. The marking process is always one step in advance, so the marking is already propagated via these links unless the two values were the same. The affix graph nodes connected via a 'first-in' link are marked as affected, see Figure 5.11. The marking propagated via the 'out' links are propagated to the other side of an affix position slice and then propagated to the affix graph nodes connected via 'first-in' links, if there are any.

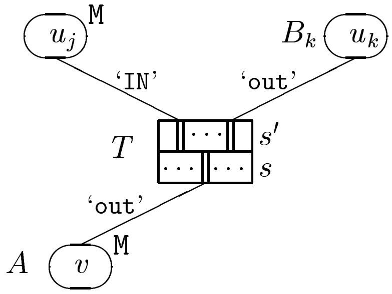


Figure 5.11: Affix position slice.

The propagation process will propagate the values via both the 'in' as well as the 'first-in' links to the affix graph nodes. However, the nodes connected to the 'in' links are not marked and the propagated value is thus only compared with the current value of such a node. The propagation process either stops in this node, if the propagated value equals the current value of the node, or starts to backtrack, if the values are different. This process is recursively repeated in each marked affix graph node until the tree-graph is consistent again or no such consistent tree-graph can be found.

If the propagation process hits on a critical affix position of a predicate, this predicate is re-evaluated and the affix graph nodes connected to the non-critical affix position slices via a 'first-in' link are marked as affected.

Shortcomings of the marking algorithm

The second solution is much more efficient than the first one, because the propagation of a value `v` can stop if this value is the same as the value of a marked affix graph node visited. In the first solution all affected affix graph nodes have to be visited twice. In spite of loss of efficiency we have selected this first solution, because it guarantees that none of the undesired effects described in the rest of this section occur.

The second algorithm operates in a depth-first manner. This is why the re-evaluation of the predicates may be problematic, especially if they have more than one critical affix position. If several of these positions of a predicate depend on the pruned subtree-graph, this predicate may need to be evaluated several times before a consistent decoration of the tree-graph is found. Although several of the critical affix positions are affected, only one of these receives a new value because of the depth-first method. To guarantee a proper re-evaluation of such a predicate, it should become delayed until all affected critical positions have obtained a new value. This is not possible unless the complete graph is inspected to see whether other critical affix positions of this predicate depend on the changed subtree-graph. However, we do not have the dependency graph explicitly available. The propagation algorithm will therefore immediately re-evaluate the predicate, and because of the potential illegal combination of values for the affix positions the outcome of the evaluation of the predicate is suspicious. Or, worse, a consistent decoration exists but is never found because the illegal combinations of values always result in a failure of the evaluation process. The possibility of several re-evaluations of the same predicate makes the efficiency gain questionable, because each re-evaluation may involve a recalculation of affix values of a considerable part of the tree-graph.

The first solution causes no problems with the re-evaluation of predicates, because all affected critical affix positions will be marked by the $\perp$-propagation algorithm and the predicate is thus only evaluated if all affix graph nodes connected to the critical affix position slices have a value again. The second advantage of this solution is that the propagation algorithm itself need not be modified, it is simply preceded by the $\perp$-propagation phase.

5.2.3 Affix value propagation in erroneous subtrees

In this section we discuss the propagation of affix values in a tree-graph which contains an erroneous subtree. If a syntax or type error is detected during the parsing of the new text, this text is transformed into an erroneous subtree. The user is not forced to correct the errors before going on with the rest of the edit actions. This can be achieved by grafting erroneous subtrees onto the tree-graph in order to enable the user to correct them later. The structure of the erroneous subtree is discussed in Section 4.2.1. The contents of the connected affix graph nodes is the $\perp$-value. In the previous section we discussed the $\perp$-propagation algorithms and remarked that they do not use backtracking. So if the parsing fails and an erroneous subtree is built no extra propagation actions are needed when this subtree-graph is grafted, because the values of all affix graph nodes depending on the new subtree-graph remain the $\perp$-value.

5.3 Ambiguity

Having found a successful parse, the parser yields a tree-graph. This tree-graph must be copied when performing edit actions on it, because in the backtrack phase the tree-graph is dismantled (Section 3.3). However, our parser does not stop after the first successful parse, so several tree-graphs may be built. Somehow all these tree-graphs must be combined into one tree-graph, because the user can only edit one tree-graph[2].

The result of combining multiple tree-graphs will be a *3-dimensional tree-graph.* Tree-graphs with distinct subtree-graphs are mapped onto one tree-graph. Corresponding subtree-graphs which are not equal can all be found in this 3-dimensional tree-graph, their root nodes are connected to a new type of tree node, the *3-dimensional* tree node.

[2]A possible solution may be that the system automatically selects one of these tree-graphs, but in that case it is possible that a tree-graph is selected which the user did not have in mind. This can be prevented by allowing the user to direct the selection, as is done in the ASF+SDF meta-environment [Kli91].

The sons of the 3-dimensional node are subtree-graphs which have the same root and represent the *same* substring in the input sentence. The internal structure of each of these subtree-graphs is in general different.

First we need the following definitions:

$$\begin{aligned} N \neq_T N' \;\Leftrightarrow\; & \exists N : \alpha. \in P,\; N' : \alpha'. \in P : (N = N' \;\wedge\; \alpha \neq \alpha')\vee \\ & (N = N' \;\wedge\; \alpha = \alpha' \wedge \\ & \exists N_i \in \alpha, N'_i \in \alpha' : N_i : \beta. \in P,\; N'_i : \beta'. \in P \;\wedge\; \beta \neq \beta') \end{aligned}$$

$$\begin{aligned} N =_T N' \;\Leftrightarrow\; & \exists N : \alpha. \in P,\; N' : \alpha'. \in P : \\ & (N = N' \;\wedge\; \alpha = \alpha' \wedge \\ & \forall N_i \in \alpha, N'_i \in \alpha' : N_i : \beta. \in P,\; N'_i : \beta'. \in P \;\wedge\; \beta = \beta') \end{aligned}$$

In both tree-graphs yielded by the parser, T_1 and T_2, there are corresponding paths from root node N_0 to node N_j such that for each pair of nodes $N_{i,1}$ and $N_{i,2}$ with $i < j$ the condition $N_{i,1} =_T N_{i,2}$ holds, and $N_{j,1} \neq_T N_{j,2}$ holds. Thus, in N_j the two tree-graphs start to diverse. In Figure 5.12 the two nodes for which different alternatives were chosen have the labels $N_{j,1}$ and $N_{j,2}$ respectively.

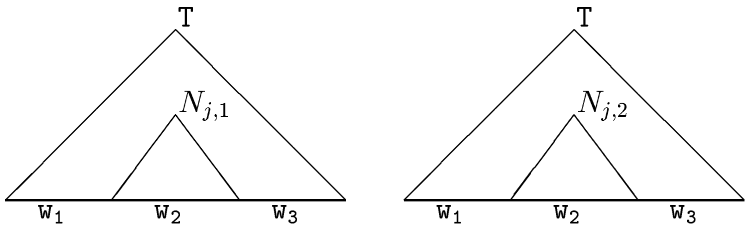


Figure 5.12: Two tree-graphs for an ambiguous sentence.

The extent of $N_{j,1}$ equals the extent of $N_{j,2}$, but they represent different alternatives for the same non-terminal in the context-free grammar. In Figure 5.13 the two tree-graphs are combined in one 3-dimensional tree-graph.

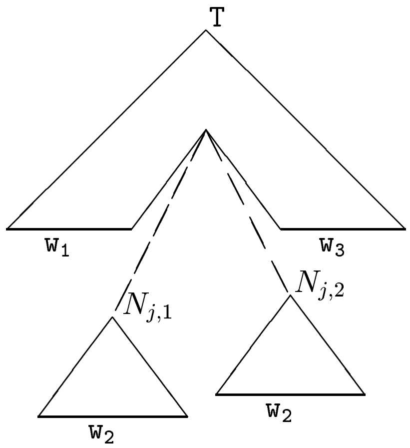


Figure 5.13: 3-dimensional tree-graph for an ambiguous sentence.

We must be careful if $N_{j,1}$ and/or $N_{j,2}$ have more than one son, because $N_{j,1}$ and $N_{j,2}$ may not have the same substring as extent. Consider the ambiguous rule:

```
exp:
  exp,
    "+",
      exp.
exp:
  id.
```

When the parser has, for instance, recognized the sentence `x+x+x`, the two tree-graphs shown in Figure 5.14 will have been built.

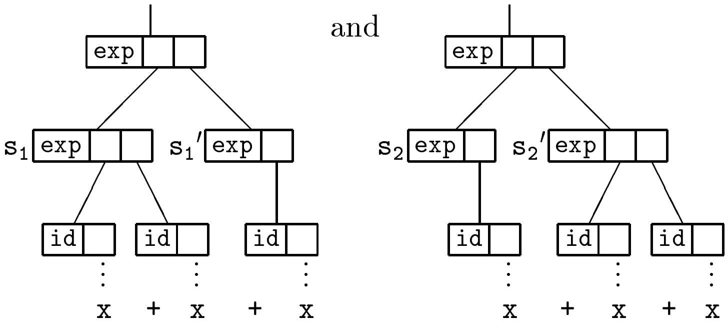


Figure 5.14: Tree-graphs for `x+x+x`.

Their naive combination is shown in Figure 5.15. The nodes `S` and `S`′ are the 3-dimensional nodes in the tree-graph. This 3-dimensional tree-graph represents three different unparsings, because of possible illegal combinations:

`x + x` viz. $s_2{+}s_1'$
`x + x + x` viz. $s_1{+}s_1'$ or $s_2{+}s_2'$
`x + x + x + x` viz. $s_1{+}s_2'$

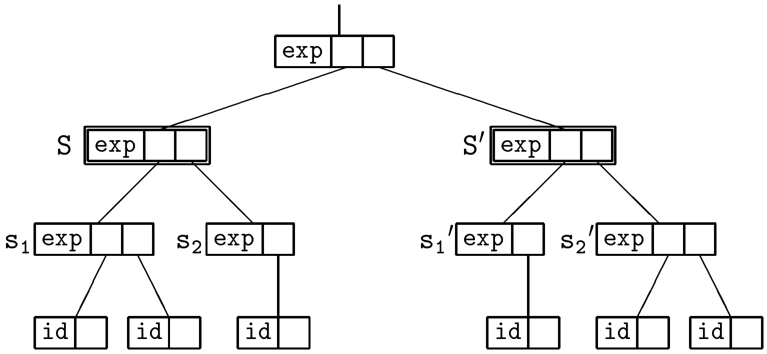


Figure 5.15: 3-dimensional tree-graph.

The illegal combinations can be prevented by combining these subtree-graphs at a node higher in the tree-graph, see Figure 5.16.

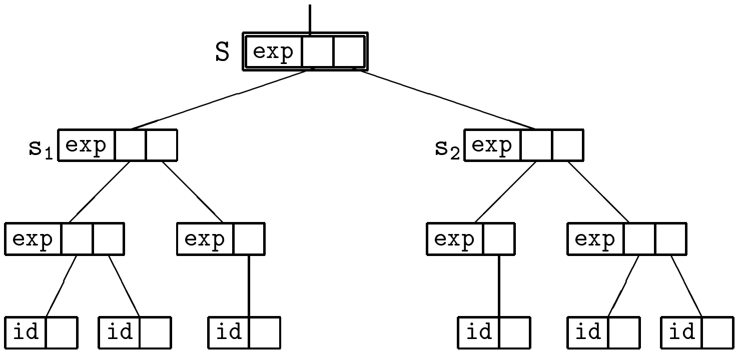


Figure 5.16: Correct 3-dimensional tree-graph.

A tree folding mechanism is invoked each time an alternative tree-graph is found for the same input sentence. Instead of being duplicated, the alternative tree-graph is compared with the already available (3-dimensional) tree-graph and a new 3-dimensional tree-graph is built.

A similar technique for folding syntax trees of ambiguous (natural language) sentences is discussed in [Hal91]. Their technique goes even further: it shares as many subtrees as possible within the subtrees of an 3-dimensional node. This technique was developed to increase the storage efficiency of the LDB-system, a Linguistic Database program [HH90].

5.3.1 Unparsing of 3-dimensional tree-graphs

The various unparsings of the tree-graphs built for an ambiguous sentence need not be the same. The editor, however, can only display *one* unparsing. The different subtree-graphs are all combined using 3-dimensional tree nodes. The substrings with different unparsings are therefore localized.

The unparse routine nevertheless traverses all subtree-graphs of a 3-dimensional node, in order to be able to select each of the subtree-graphs built for an ambiguous sentence so links to all individual subtree-graphs must still be stored.

5.3.2 Affix value propagation in 3-dimensional subtree-graphs

The decoration of the corresponding affix graph nodes is in general different in each of the tree-graphs yielded by the parser. The tree-graphs yielded are transformed into one tree-graph and the resulting affix graph nodes must again become consistent.

Each successful recognition of the same sentence results in a decorated tree-graph. The tree-graphs are compared one by one and equivalent parts are mapped onto each other. Differing subtree-graphs are combined into 3-dimensional subtree-graphs. During this process only tree nodes are compared, not the contents of the affix graph nodes.

The contents of the affix graph nodes may also be different. There are several ways of solving this problem.

1. The 3-dimensional node is introduced as soon as equivalent affix graph nodes in different tree-graphs built for the same input sentence have different values. The consequence may be that the amount of sharing in tree-graphs is very restricted.

2. The contents of the resulting affix graph node becomes the $\Box$-value as soon as equivalent affix graph nodes in different tree-graphs have different values. This yields no problems because each parse was syntactically and static semantically correct.

3. Both values are stored in the same resulting affix graph node.

4. The values of the affix graph nodes of only the first parse are stored and the values of the affix graph nodes of the other parses are ignored.

We have opted for the second solution, which works fine.

5.3.3 Incrementality and ambiguity

Fully reparsing the sentence is inevitable for an ambiguous context-free grammar in order to guarantee that the user gets all information. So an attempt to maximize the incrementality is superfluous. Each edit action may introduce or delete several parsings.

Although ambiguity in general renders incremental techniques useless, the techniques introduced above are still useful to tackle the problem of untyped placeholders, since these cause 'local' ambiguities.

5.4 Placeholders

Placeholders and templates play an important role in PREGMATIC. Both typed and untyped placeholders as well as templates are automatically derived from the specification as described in Chapter 4. The usefulness of, and extra convenience introduced by the untyped placeholders has been explained in Section 1.4. In this section we discuss the consequences for the parser and affix value propagation.

An untyped placeholder can be replaced by several typed ones, where each of the substitutions results in a successful parse and a corresponding tree-graph. These tree-graphs can be combined and result in a 3-dimensional tree-graph, as discussed in Section 5.3.

Untyped placeholders may represent all non-terminals in the language except semi-terminals.

It is necessary to transform untyped placeholders into typed placeholders in order to be able to work with them. All rules of the grammar are implicitly extended with an extra alternative, and as a result the grammar becomes ambiguous. The following PICO program contains an untyped placeholder:

```
BEGIN DECLARE x; x := <|> END
```

The (only) possible replacements for <|> in this program are:

- <|expression|>

- <|term|>

- <|factor|>
- <|identifier|>
- <|number|>

The main problem is the assignment of non-terminal names to the untyped placeholders during parsing. Each parse-routine get_N is extended with two extra alternatives, for the recognition of the typed and the untyped placeholder respectively.

```
get_N()
{
  push_q(<<red_N()>>);
  push_q(<<sym_symbol("<|N|>")>>);
  push_q(<<make_tree_node("N")>>);
  call_q();
  pop_q(3);
  push_q(<<red_N()>>);
  push_q(<<sym_symbol("<|>")>>);
  push_q(<<make_tree_node("N")>>);
  call_q();
  pop_q(3);
  push_q(<<red_N()>>);
  ⋮
}
```

An untyped placeholder is replaced by a typed one in each tree-graph yielded by the parser but these typed placeholders need not be the same. If an untyped placeholder can only be replaced by *one* typed placeholder no 3-dimensional subtree-graph will be built and the textual representation of this untyped placeholder is automatically changed into the typed one.

The unparsing of an untyped placeholder which represents several typed ones is still the untyped placeholder symbol. It is connected to all typed replacements internally during unparsing.

All possible typed replacements are shown, together with all possible templates (Figure 5.17) when the user focuses on an untyped placeholder. A selection causes the replacement of an untyped placeholder by a typed one, or by a template. The structure of the 3-dimensional tree-graph may also be changed, but the reparsing mechanism will take care of that, by starting in the highest 3-dimensional tree node, see the next section. It is therefore possible that subtree-graphs and/or 3-dimensional tree nodes disappear as a result of an edit action.

The introduction of untyped placeholders causes the grammar to become (locally) ambiguous. In Section 5.3.3 it was remarked that reparsing should start in the root node in order to ensure that all information is available. Because of the local nature of the ambiguity, the reparsing of a subtree-graph which contains an untyped placeholder always starts in the highest 3-dimensional tree node.

If inserted text contains a placeholder no extra attention is paid to it. If parsing has succeeded no attempts to find more typed replacements by trying higher nodes are made.

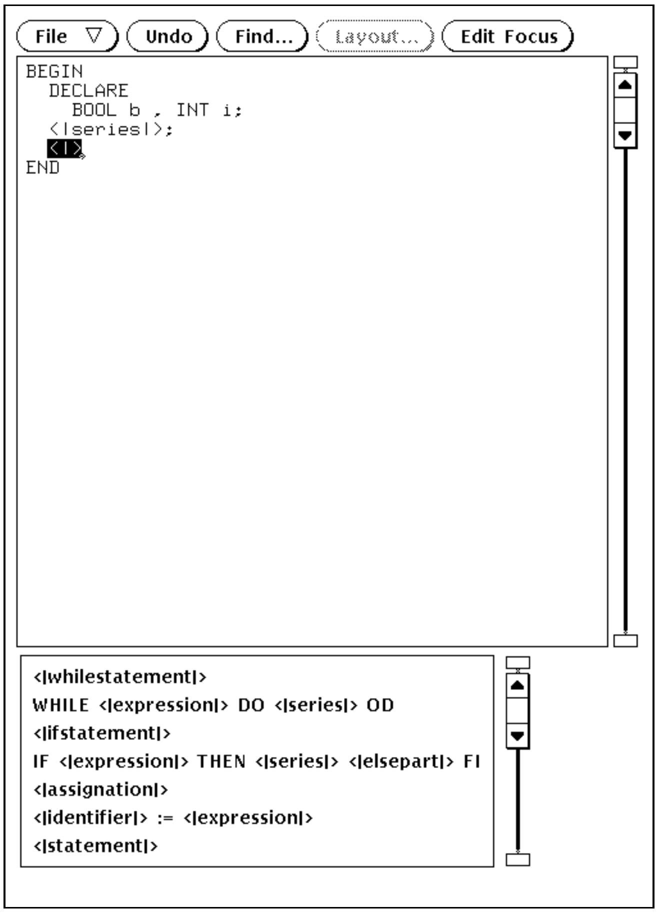


Figure 5.17: Untyped placeholder focused.

5.4.1 Templates

If the user focuses on a placeholder, the template window shows all possible syntactically correct templates. If the user clicks on a template the tree-graph has to be adapted. We have chosen the template starting in the focused (typed) placeholder or in the highest 3-dimensional tree node for reparsing.

The use of a prefabricated subtree-graph for templates would not work for a replacement of an untyped placeholder, because the selection of the template may initiate the restructuring of the 3-dimensional tree-graph.

5.4.2 $\square$-value propagation

The $\square$-values are strongly related to placeholders (Figure 5.18). These values enable us to do just as much type checking as is possible when the program contains placeholders.

The $\square$-value is propagated as an ordinary affix value, but if it is assigned to an affix graph node which already contains a value $\mathtt{v} \neq \square$, the affix value propagation process stops propagating the □-value, and starts propagating value v. This value is propagated 'backwards' via the links over which the □-value was propagated.

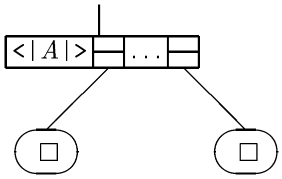


Figure 5.18: Placeholder node for non-terminal A.

There is some resemblance between the □-value propagation mechanism and unification. In a unification mechanism the propagation of a more defined set of values is preferred over the propagation of a less defined set of values. There is no other resemblance, because the evaluation mechanism always propagates only one value instead of sets of values.

5.5 Unparsing

One of the remaining topics we want to address is incremental unparsing. Although the introduction of the user-interface based on the X-window system has made incremental unparsing less urgent. IO-performance in this user-interface even improved using a non-incremental technique. We will nevertheless briefly discuss some of the results obtained by applying this technique.

The unparsing algorithm has to traverse the tree-graph twice. The first pass calculates the length of each syntactical construct, and the second pass writes the terminal symbols and layout symbols to the screen buffer. In the first pass only the length of those tree nodes affected need be recalculated. This incremental feature is still used.

The incremental optimalizations in the second pass are rather restricted. It is not sufficient to visit only the tree nodes affected during the unparsing of the tree-graph, because the effect of the edit action may affect the unparsing of tree nodes which themselves are not affected. This is the case in recursive rules where a non-recursive member is changed, and the effect of this alteration is that the orientation of the unparsing of this rule is changed.

All edit operations affect the contents of the screen buffer. Unparsing the tree-graph can be considered as inserting and deleting layout symbols in the screen buffer, which can be done in such a way that a only minimal number of screen updates are necessary. Applying this principle in combination with the X-window system causes annoyingly slow window updates.

5.6 Efficiency considerations

We noted at several places that the efficiency of the total system is not optimal, but in the introduction we stated that we prefer generality to efficiency. The use of the backtracking mechanism may lead to an exponential behaviour of the parser and the type checker. Much of the inefficiency in the parser can be prevented by examining the eag carefully, and by transforming some of its rules.

In this section we will describe a few improvements to the system which may lead to a better performance. We will also give a short list of hints for improving the eag.

5.6.1 Re-use

The reparsing of the changed extent of a node may involve unnecessary work when parts of this extent have already been parsed in previous parses. The parsing of these parts may yield the same subtree-graphs as the time before. We could enlarge the incrementality by re-using these subtree-graphs rather than parsing the extent of these tree-graphs.

Each character of the extent of the root node of the tree-graph is linked to a tree node (Section 4.1.1). The links created by the unparsing algorithm are used by the focus mechanism. They can also be used by the reparsing algorithm described in this section.

Immediately after an edit action, all tree nodes on the path from the focus to the root are marked as *affected*. The subtree-graphs without affected roots may be re-used by the reparsing algorithm. We want to use as much as possible of the previous tree-graph, but we cannot prevent the reparsing of new text and of pieces of the extents of affected nodes.

This technique for increasing the incrementality is particularly fruitful if the complete tree-graph must be reparsed, but it can also be used during the reparsing of text yielded by the text edit window. The implementation of the parser is discussed in Section 3.1.3. The recognition of text derivable from a non-terminal N is done by the parse-routine `get_`N. This routine is extended with a mechanism to scan the previous tree-graph for a node N of which the extent starts at exactly the current input position and which is not marked as affected. If such a node is found, instead of parsing the extent of this node, the corresponding subtree-graph is retrieved and the input position is set to the position immediately after the extent of this node. The modified parse-routine `get_units` from Section 3.1.4 looks like:

```
get_units()
{
  node := lookup_in_tree("units");
  if (node <> nil) {
    push_q(<<red_units()>>);
    push_q(<<retrieve_subtree(node)>>);
    call_q();
    pop_q(2);
  }
  else {
    push_q(<<red_unit()>>);
    ⋮
  }
}
```

The routine `lookup_in_tree` will return the tree node if it is found and it satisfies all conditions, i.e. it may not be marked as affected; otherwise it will return `nil`, representing the empty tree node. The routine `retrieve_subtree(node)` pushes the tree node `node` on the tree node stack and moves the input position beyond the extent of `node`; these actions are reversed during backtracking. The corresponding affix graph is of course also copied.

This technique is also used in reparsing the text yielded by the text edit window. Each character of the text in the text edit window is still linked to the subtree-graph of the focus. During this edit action the link administration of the extent of the focus is continuously updated. Newly inserted characters are not linked to the tree-graph. Deletion of characters also causes deletion of the corresponding links to the tree-graph. Clicking on the `put back`-button causes the replacement of the extent of the focus and an update of the complete link administration. The reparsing mechanism may use this information during the parsing of the modified extent.

5.6.2 Re-using results of predicate evaluations

Another improvement is concerned with the evaluation and re-evaluation of predicates. In the current implementation each predicate occurrence is evaluated, independently of other occurrences of the same predicate. Such a predicate may be evaluated several times during the decoration of the affix graph and it may be re-evaluated several times with the same values for its critical affix positions. Re-evaluation of a predicate with the same values for the critical affix positions will always have the same result. These (re-)evaluations can be improved to obtain more efficient affix value propagation.

In order to be able to do this we must ensure that affix values are mapped onto each other, two affixes with the same value refer to the same memory location containing this value. This is not a major modification in the system, each time an affix value is created it must be checked whether this value already exists. If the system satisfies this constraint we are able to improve the evaluation of the predicates.

The main idea is that no predicates are evaluated twice for the same values of the critical affix positions during an edit session, the complete sequence of edit actions between starting up and shutting down the system. This can be achieved by storing (caching) the values of the critical affix positions and the result of the evaluation and/or the resulting values of the non-critical affix positions for each predicate. Each time a predicate is evaluated this list is first inspected, if the predicate has not yet been evaluated for these values the predicate must be evaluated and the results stored. If the predicate has already been evaluated the result of the previous evaluation are re-used. This strategy is possible because the evaluation of predicates is deterministic. The performance can be improved even further by using a hashing technique. A similar technique, hash-consing for tree-graphs, is used in HAG [VSK91], to improve the evaluation of non-terminal attributes (NTA).

5.6.3 Inconsistencies detected after grafting subtree-graphs

One of the weak points of our system with respect to incrementality is that an inconsistency detected during the affix evaluation started after grafting a syntactical and static semantical correct subtree-graph results in a destruction of this grafted subtree-graph. An erroneous subtree is built and grafted which contains the extent of the previously built subtree-graph but has no internal tree structure. At this point the performance can be improved by not backtracking over the grafting action, but by reporting only the error.

This strategy could be generalized, but then the joint-venture of the parser and affix value propagation has to be adapted. The detection of a type error does not cause backtracking, but allows the parsing process to continue. If the syntactical recognition succeeds the tree-graph is completed with as much type checking information as possible. We performed some experiments using this strategy and were not really satisfied with it. There was a high degree of ambiguity since for each failing alternative an attempt was made to recognize the rest of the sentence which quite often succeeded. We tried to improve this by formulating stronger conditions to continuing the parsing process but this did not lead to satisfactory improvements in performance. This generalization has therefore not been implemented in the prototype.

5.6.4 Improvements

The performance of the system can also be improved by transforming the eag specification:

1. left-factorization of the underlying context-free grammar.
2. application of predicates as soon as possible. Try however to avoid unnecessary delaying.
3. use of free affix non-terminals if an affix value is only transferred in an alternative. This prevents unnecessary checks.

This list of improvements is not exhaustive, but may suggest other transformations.

Chapter 6

Execution

Until now we have discussed topics which are related to syntax-directed editors, but which were not specific to programming environments in general. One way to turn a syntax-directed editor into a programming environment is by adding a component for executing the developed programs, implemented as:

- an interpreter,
- a compiler, or
- a debugger.

We describe only an interpreter-based approach, because the implementation of a debugger is very much like that of an interpreter. A compiler could be generated using the compiler generator Programmar [Mei86].

We will describe three different methods which could be used for implementing an interpreter in PREGMATIC:

1. evaluation by means of intermediate code.
2. evaluation by means of predicates.
3. evaluation by means of evaluating affix expressions.

We have not yet implemented any of these methods in the prototype of PREGMATIC, but we did experiment to some extent with the second and third method. All three models make it necessary to adapt the EAG-formalism which must be extended with facilities which allow the specification writer to mark either the affix positions or the predicates used for the dynamic semantics. We have not yet decided what these extensions will look like and therefore we could not implement the interpreter in the prototype.

The user-interface needs to be extended with an execute-button which opens an IO-window in which the user may interrupt, resume, or restart execution.

The three models will be demonstrated by using the same example. We will give the specification of the dynamic semantics of the following syntactical construct.

```
whilestatement:
  "while",
    expression,
      "do",
        series,
          "od".
```

We discuss the three possible implementation models in the next three sections and look at IO, incrementality, adaptations to the EAG-formalism, and relation to other systems. We conclude this chapter with an overview of the techniques used in other systems.

Incrementality again plays an important rôle in the execution of programs. The first view of incremental execution is that an arbitrary piece of program may be selected and executed [Kai89] provided it is complete and consistent. This strategy is applied in the ELAN-programming environment [KW86]. If this selected piece of program depends on other parts of the program, for example for initialization of variables, the interpreter will immediately report an error. The second view is that incomplete programs, viz. programs containing placeholders may be executed. The third view is that as much information from previous interpretation sessions of the same program is re-used [WJ88] as possible.

6.1 Code generation

In this section we present the first method of implementing an interpreter in PREGMATIC. The method is straightforward: translating the developed program into intermediate code and executing that code.

During the development of the program a list of instructions is generated in some intermediate language using the normal affix evaluation mechanism. These instructions are interpreted by a language independent interpreter. This interpreter only has knowledge of the intermediate code. A similar technique is used in the Synthesizer Generator [RT89a].

Each syntactical construct is described by a number of instructions. The affix evaluation mechanism takes care of constructing the complete list of instructions representing the translation of the developed program to intermediate code. The representation of the constructed list depends on arbitrary design decisions, which may effect the incremental behaviour of the system. Some implementations are better suited to incremental generation of the list than others.

We will demonstrate this execution strategy by means of an eag which generates code for a stack-oriented interpreter. The instructions consist of a label followed by an opcode and zero or more operands. Values are pushed onto the stack and the operations modify the contents of this stack. The rule for the `whilestatement` is given below. The non-terminals are extended with affixes needed for the generation of the code. We have left out the affixes describing the type checking.

```
whilestatement (nextlabel, expressionlabel,
               expressioncode +
               boclabel + ":" +
```

```
              "BOC(" + serieslabel + "," + nextlabel + ")" + nlcr +
              seriescode +
              gotolabel + ":  GOTO " + expressionlabel + nlcr):
  "while",
    make label (boclabel),
      expression (boclabel, expressionlabel, expressioncode),
        "do",
          make label (gotolabel),
            series (gotolabel, serieslabel, seriescode),
              "od".
```

BOC stands for BranchOnCondition. The call make label is a call to a predicate which will generate a unique label, the exact implementation details of this predicate are not relevant to understanding this example. The affix non-terminals nextlabel and expressionlabel contain the labels of the instruction following the whilestatement and of the first instruction of the whilestatement respectively.

We will only give the code for the following piece of program:

```
while
  x /= 9
do
  x := x + 1
od
```

The resulting list of instructions for this program fragment will be:

```
 ⋮
 7: PUSHVAR  x
 8: PUSHVAL  9
 9: NOTEQUAL
10: BOC      11,16
11: PUSHVAR  x
12: PUSHVAL  1
13: ADD
14: STORE    x
15: GOTO     7
16: ...
```

This method may be the most tedious one for the specification writer because he may also have to write the interpreter. This need only be done once. The dynamic semantics of other languages can be described using the same interpreter.

6.1.1 IO-facilities

The specification of the complete dynamics of a language asks for facilities for reading and writing of values — IO-facilities. The specification writer has to know which facilities are offered by the interpreter. They may be very different, ranging from very simple routines for reading and writing a single character to complex routines for window manipulations.

6.1.2 Incrementality

The incrementality of this type of interpreter will be very restricted. The current implementation of the affix evaluation mechanism would even make execution impossible so long the program contained placeholders. No code will be generated if the construction of the list of instructions is done by using the concatenation operator. This is of course undesirable but correcting it would mean a complete re-implementation of the evaluator. It is also possible to construct the list of instructions using the tuple operator. The resulting representation will not be a list but a kind of tree structure, which can easily be transformed into a list representation before it is passed to the interpreter.

List representation of the code makes it very difficult to execute only part of the program. Furthermore, a small alteration of the program text involves a complete reconstruction of the generated list of instructions. Processing dynamic semantic information in environments generated by the Synthesizer Generator [RT89a] does not suffer from the two problems described above. The evaluator in the environments generated by the Synthesizer Generator [RT89a] stores the code generated for each syntactical construct in an attribute and these attributes are linked, forming a code graph for the developed program. The interpreter traverses this graph by following the links and executes the code stored in each attribute node. The advantage of this strategy is the limited amount of recomputation after an alteration of the program text and that it enables the user to execute only a part of the program.

A similar solution could also be used in the environments generated by PREGMATIC. This would mean an extension of both the EAG-formalism and the affix evaluation mechanism. The affixes describing dynamic semantic information have to be marked in some way. The evaluation mechanism needs to be extended with a mechanism to directly link the affix graph nodes containing this dynamic semantic information to each other.

Re-using as much information as possible of previous executions from the same program in this interpretation model is only possible if the interpreter is adapted to keep track of values which can be re-used from previous calculations.

6.2 Interpretation within EAGs

In this section we describe the second method of tackling the problems related to the execution of programs developed in PREGMATIC. In the previous section we worked with a language independent interpreter and intermediate code. The approach discussed in this section is also based on intermediate code. The interpreter is not language independent, but defined as a predicate in the specification. The intermediate code generated is a more abstract representation of the program developed and will be interpreted by that predicate. The predicate resembles the ones used for the specification of the type checking rules.

The variables and their values are stored in a so-called value environment, an abstract representation of the memory locations used by an ordinary interpreter. The right hand side of the rule for `program` is extended with a call to the `interpret`-predicate. Again the eag does not contain any type checking information. The non-terminal `series` in the right hand side of the rule for `program` therefore has only one affix non-terminal. This non-terminal `code` contains the abstract representation of the recognized program.

```
program:
  "begin",
    declarations,
      series (code),
        "end",
          interpret (code, nil, value-env).
```

We will only give the rule for `whilestatement`.

```
whilestatement ("while"*cond*loop):
  "while",
    expression ("bool", cond),
      "do",
        series (loop),
          "od".
```

The abstract representations of `expression` and `series` are stored in the affixes `cond` and `loop` respectively, which are combined and extended with a label `while`, to indicate that a loop should be executed and propagated as a 'synthesized' affix.

We will now only give the alternative for the `whilestatement` from the specification of the predicate `interpret` and the definition of the predicate `interpretwhile`.

```
interpret (>"while"*expr*while, >old values, new values):
  interpretexpr (expr, old values, val),
    interpretwhile (val, expr, while, old values, new values).

interpretwhile (>"true", >expr, >while, >old values, new values):
  interpret (while, old values, values),
    interpretexpr (expr, values, val),
      interpretwhile (val, expr, while, values, new values).
interpretwhile (>"false", >expr, >while, >values, values):
  .
```

This method of executing programs depends heavily on predicates. The calculation of the values of the variables must be specified by means of predicates. A lot of primitive predicates will be necessary to make these calculations possible, or to make them more efficient. A primitive predicate for performing multiplication will be more efficient than specifying the multiplication by means of repeated additions. There are also other shortcomings related to this solution which will be discussed in the next sections.

6.2.1 IO-facilities

Specification of IO in this method is also based on primitive facilities. The capabilities of these facilities are comparable to those of the IO-facilities of the previous method for executing programs the representation of the IO-channels, which take care of the transfer of values from and to the output device, either a file or a window, may however be problematic. In the previous method these channels were implicitly available via the interpreter, now they have to be explicitly specified as affixes.

The first solution is to represent the IO-channels as ordinary affixes. Almost every non-terminal in the `interpret`-predicate is extended with two affix non-terminals `in` and `out`, which represent the input channel and the output channel respectively. For example, the alternative for `whilestatement` in the `interpret`-predicate will be:

```
interpret (in, out, >"while"*expr*while, >old values, new values):
  interpretexpr (expr, old values, val),
    interpretwhile (in, out, val, expr, while, old values, new values).
```

The primitive predicates for accessing these channels are `read(in,v)`, for reading values from the input channel `in`, and `write(v,out)` for writing values to the output channel `out`. The affix position represented by `in` will be critical in the definition of the predicate `read` whereas `v` will be the critical affix position in the definition of the predicate `write`.

The disadvantage of this solution is the 'dragging' of the IO-channels through the specification of the `interpret`-predicate. This problem could be prevented by making the affixes for the IO-channels global. This means that these two affixes are always implicitly available and can be accessed at any moment via the predicates `read` and `write`. This representation of the channels would be very exceptional, because neither the specification formalism nor the system support this type of affixes. We therefore prefer the solution which 'drags' the IO-channels through the eag, although it involves some extra work.

6.2.2 Evaluation

Looking at the dynamic semantics in the EAG specification suggests that the execution of a program may be done by using the same affix value propagation mechanism as for type checking the language. Though this claim is true the specification of the dynamic semantics may yield a non-well-formed eag. The execution of a non-terminating loop yields a non-terminating affix evaluation process, which is correct with respect to the execution but not with respect to the evaluation process. The integration of parsing and affix value propagation now causes a severe problem. The evaluation of the `interpret`-predicate, which is not guaranteed to terminate, may cause non-termination of the entire parsing process.

The solution of Section 6.1 does not have these problems because the interpreter is explicitly started by the user of the programming environment, while the affix value propagation mechanism will cause the evaluation of the program implicitly. Explicit invocation of the `interpret`-predicate is also the solution for the problem described here. The construction of an abstract representation of the program may be done during the parsing of the sentence, the predicate interpret should however remain delayed until the user of the environment pushes the execute-button.

The generator of the environment should be able to recognize which predicates in the specification are used for execution of the program and which are used for type checking. This cannot be determined statically, so the EAG-formalism *needs* to be extended with an annotation for marking the predicates which specify execution.

The same affix value propagation mechanism can be used for the evaluation of the interpret-predicates. The system however has facilities to interrupt the execution of a program, by for example pushing the interrupt-button in the execute-window. The propagation process should therefore regularly inspect whether execution should be interrupted.

6.2.3 Incrementality

The specification of the dynamic semantics as given in the beginning of this section does not offer facilities for incremental execution. The interpret-predicate cannot evaluate only a part of the abstract representation. This problem can easily be solved by extending each rule with a call to the interpret-predicate. Focusing on a program fragment and invoking the interpreter will cause the evaluation of this fragment only. The presence of the extra calls to the interpret-predicate does not influence the evaluation mechanism, because these calls are also ignored by the affix value propagation mechanism if these predicates are annotated. The only disadvantage of this method is that the IO-affixes are needed in all rules. The rule for the whilestatement will be:

```
whilestatement (in, out, "while"*cond*loop):
  "while",
    expression ("bool", cond),
      "do",
        series (in, out, loop),
          "od",
            interpret (in, out, "while"*cond*loop, nil, valenv).
```

The placeholders, another facility to increase the incremental behaviour cause no problems. The placeholder will be represented as a □-value in the abstract code and will cause termination of the execution process. Using this strategy makes it difficult to focus on the placeholder which caused the termination of the evaluation process.

Re-using as much information as possible from previous executions of the same program comes more or less for free, because predicates are only evaluated once for the same affix values. An adaptation of the program text causes an alteration of the abstract code generated during parsing. Pieces of program which are not affected will generate exactly the same abstract code. Incremental re-evaluation of predicates will ensure that for these pieces of program the predicates for interpretation behave in the same way — if, of course, the alteration did not affect the value environment built during execution.

6.3 Graph-visit interpretation

The last strategy for executing programs is based on a graph-visiting evaluation mechanism. This interpretation method is essentially different from the one used in the Synthesizer Generator[RT89a], which also works with a graph, but it is very similar to the method described in [WJ88]. This interpretation technique consists of an affix evaluation mechanism which walks over the affix graph and updates affix graph nodes until a consistent decoration is found, which corresponds to the execution of the program.

The value environment is an argument of the `interpret`-predicate in the previous solution for executing programs. In this solution the value environment is represented by a well-defined set of the affix graph nodes which is updated during evaluation.

The rule for `whilestatement` describing the execution looks like:

```
whilestatement (oldvals, newvals):
  "while",
    expression (oldvals, exprval),
      loop (exprval, oldvals, newvals, loopvals),
        "do",
          series (loopvals, oldvals),
            "od".
```

where:

```
loop (>"true", >oldvals, vals, oldvals):
  .
loop (>"false", >oldvals, oldvals, vals):
  .
```

This way of interpreting programs also causes some problems. The IO-facilities used in this technique are equivalent to the facilities discussed in Section 6.2.1.

6.3.1 Evaluation

A careful examination of both the rule given above and Figure 6.1 will reveal that the affix value propagation mechanism either will stop too soon, because the consistent substitution constraint cannot be fulfilled with respect to the affix non-terminal `oldvals`, or will not stop at all, because `oldvals` does not change and thus the first alternative of predicate `loop` will always succeed.

We assume that the value environment changes during execution of the loop-body, so the value of the affix `oldvals` of the non-terminal `series` differs from the value of the affix `oldvals` of the non-terminal `whilestatement`. Therefore the propagation process starts to backtrack after executing the loop-body for the first time. This time the problems are not only caused by starting the affix value propagation mechanism too soon, viz. during the parsing and calculation of the static semantics, but they also have a more structural nature.

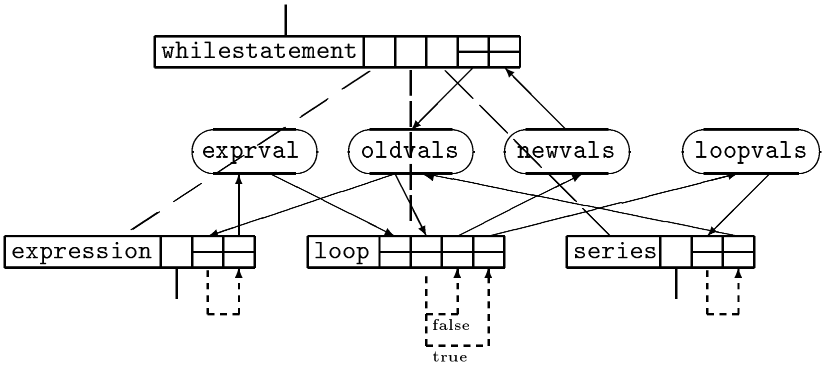


Figure 6.1: Part of a tree-graph.

The solution of the problem of mixing up the various types of calculations is solved by invoking the propagation process for the calculation of the dynamic semantics explicitly by pushing the `execute`-button. The problem of the violation of the consistent substitution constraint can only be solved by using a different propagation mechanism which does not take this constraint into consideration. If the current value of an affix graph node differs from the propagated value the current value is overwritten by the propagated one and the process must continue. An adapted version of the propagation technique such as described in Section 5.2.2 will for example do the job. One of the adaptations is that the affix link types 'first in' and 'in' should be ignored and that the propagation process must know by way of which link the previous value was propagated. The efficiency of this evaluation process can be improved by re-introducing the flow. It is otherwise possible that the wrong link will be selected to propagate the value, which could lead to a considerable amount of unnecessary work. This unnecessary work will be involved for example if the propagation process first propagates the value to the `series` non-terminal.

The affixes describing the dynamic semantics in the eag should be marked in order to choose the right propagation technique for decorating the affix graph nodes. This is impossible, so the specification formalism should be extended in order to enable the specification writer to indicate which affixes describe the dynamic semantics of the specified language. This corresponds to marking the gate attributes in the method described by [WJ88].

6.3.2 Incrementality

The incrementality in this solution is obtained for free, because the evaluation process can very easily be restricted to the subgraph related to the focused subtree-graph. Pushing the `execute`-button will only execute the extent of the focus. We will not give the exact implementation details. The presence of placeholders also causes no problems, because propagating an affix value to an affix graph node connected to a placeholder will suspend the propagation process and adjust the focus to this placeholder. This is only possible because of the direct links between the affix graph nodes and the tree nodes. The propagation process recognizes these affix graph nodes because of the □-value.

Re-using as much information as possible from previous executions of the same program is also for free, because the pieces of program which are not changed will preserve the same values for the corresponding affix graph nodes, unless these affix graph nodes depend indirectly on the changed text. The $\perp$-propagation algorithm will also mark those affix graph nodes containing dynamic semantic information affected.

Chapter 7

Case-study: SASL

Programming environments can in fact be generated for all kinds of programming languages and specification languages. We initially considered Pascal [JW86] but this language has already been used extensively for this purpose. Several specifications exist in different formalisms, SSL [RT89b] and ASF+SDF [Deu91]. Specifications for Pascal in the EAG-formalism can be found in [Wat74, Wat79].

A possibly more interesting language we considered was the functional language Miranda [Tur90]. This language has a number of very interesting properties:

- offside rule,
- expression oriented ('everything is an expression'),
- application before definition, and
- severe type checking rules.

Although a lot of work has already been done [Hie87], a full specification in EAG would still involve too much work. Still fascinated by functional languages and their specification problems we rediscovered the predecessor of Miranda: the language SASL [Tur79]. This language has almost the same interesting problems as Miranda except for the type checking. The language is not as large and therefore better suited as a case-study.

SASL stands for "St. Andrews Static Language". It is a mathematical notation for describing certain kinds of data structures. It contains no commands and a data structure, once defined, cannot be altered [Tur79]. We will not discuss all features of the language. We restrict ourselves to those which are relevant to the EAG specification of SASL.

7.1 Syntax of SASL

We first give a context-free grammar describing the syntax of SASL in BNF. Terminals are written in uppercase letters and they are surrounded by quotes, for example `"WHERE"`, `";"`, etc. Non-terminals are written in lowercase letters.

```
expr → expr "WHERE" defs |
        condexp
condexp → opexp "->" condexp ";" condexp |
           listexp
listexp → opexp "," listexp |
           opexp "," |
           opexp
opexp → prefix opexp |
         opexp infix opexp |
         comb
comb → comb simple |
        simple
simple → name |
          constant |
          "(" expr ")"

defs → clause ";" defs |
        clause
clause → namelist "=" expr |
          name rhs
rhs → formal rhs |
       formal "=" rhs
namelist → struct "," namelist |
            struct "," |
            struct
struct → formal ":" struct |
          formal
formal → name |
          constant |
          "(" namelist ")"

constant → numeral |
            charconst |
            boolconst |
            "()" |
            string
numeral → real scalefactor |
           "-" real scalefactor
real → digit+ |
        digit+ "." digit+
scalefactor → "e" digit+ |
               "e" "-" digit+ |
               ε
boolconst → "TRUE" | "FALSE"
charconst → "%" anychar | "SP" | "NL" | "NP" | "TAB"

prefix → "~" | "+" | "-"
infix → ":" | "++" | "|" | "&" | ">>" | ">" | ">=" | "=" | "~=" | "<=" | "<" |
         "<<" | "+" | "-" | "*" | "/" | "DIV" | "REM" | "**" | "."
```

The notion `digit`$^+$ stands for one or more digits. The last alternative of `scalefactor` represents the empty alternative.

All infix operators are left associative except for the operators ":","++","**", and ".". The priority rules for the operators are:

{:,++}[1] < {|} < {&} < {~} < {>>,>,>=,=,~=,<=,<,<<} <
{+,-}[2] < {+,-}[3] < {*,/,DIV,REM} < {**,.}[1]

Some of the non-terminals in the context-free grammar of SASL are not defined. These non-terminals represent the terminal symbols `name`, `anychar`, `digit`, and `string`. A `name` is any sequence of lowercase letters, digits and the symbol "_" which starts with a lowercase letter. The hyper rule for `name` is:

```
name:
  {abcdefghijklmnopqrstuvwxyz} (l),
    {abcdefghijklmnopqrstuvwxyz0123456789_}*!  (ls).
```

The hyper rule for `digit` is:

```
digit:
  {0123456789} (d).
```

The non-terminal `anychar` represents any visible character. The non-terminal `string` stands for an arbitrary number of characters prefixed by a single quote character and postfixed by a double quote character.

Layout in SASL consists of spaces, newlines, and comments. Layout is always optional except when leaving out the layout between two symbols would lead to a new name. For example, leaving out the layout in the fragment `f 3` would result in `f3`, which is a new name instead of a name followed by a constant. SASL obeys the *offside rule*, which will be explained in Section 7.3. We have omitted SASL comments.

7.2 Semantics of SASL

Type checking in SASL is in fact very simple. It consists simply of the identification of identifiers. There is no check on types of operands or the arguments of functions. If an operator is called with inappropriate types the result of the evaluation will be the `undefined` object.

There are 6 types of objects in SASL, which correspond to the data types of expressions:

1. `numbers`, which may be positive, negative, or zero.

2. `truthvalues`, which are represented by the values `TRUE` and `FALSE`.

3. `characters`, which are always preceded by a "%"-character, for example `%B`, `%c`, `%%`, and the unprintable characters: `SP`, `NL`, `NP`, and `TAB`.

---
[1]Right associative.
[2]The dyadic infix operators + and -.
[3]The monadic prefix operators + and -.

4. `lists`, which are ordered sets of objects, these objects are called elements.

5. `functions`, which assign an output object to an input object of the right type. This output object is the result value of the evaluation of the function for the input object.

6. `undefined`, which represents expressions which are not well-formed or do not terminate, for example `3 + TRUE`, or `fac -3`.

The following holds for all types of objects:

- Any object can be named.
- Any object can be the value of an expression.
- Any object can be an element of a list.
- Any object can be given to a function as its input.
- Any object can be returned by a function as its output.

Objects of different types may be freely mixed, for example

```
a > b -> TRUE ; 7
```

Even elements of lists need not be of the same type:

```
1, TRUE, 3 + TRUE, (1, 2, 3)
```

It is impossible to have a static type check mechanism with these type checking constraints.

7.3 Offside rule

While the type checking constraints are not really interesting, SASL has another interesting feature, viz. the offside rule. Describing this feature in the EAG-formalism is not trivial. The offside rule in the SASL-manual [Tur79] is formulated as

> Every symbol of an expression must lie below or to the right of the first symbol of the expression.

The specification of the offside rule in the EAG-formalism is based on a modified version of this rule. This version was also implemented in the SASL system under UNIX.

1. In a definition, represented by the non-terminal `clause`, every token of the `expr` following the `"="`-symbol must be to the right of the column containing this `"="`-symbol.

2. In a conditional expression, represented by the non-terminal `condexp`, every token of the `expr` following the "->"-symbol must to the right of the column containing the "-" of this "->"-symbol.

The offside rule is meant as an implicit delimiter, as an alternative to the delimiter ";" in the following construction:

```
expr WHERE clause ; defs
```

In this case the `defs` need not obey the offside rule but no token belonging to `clause` may be offside. The following SASL fragment, for instance, is written using the offside rule.

```
f 3
WHERE f x = g y z
            WHERE y = ( x + 1 ) * ( x - 1 )
                  z = x ** 2 + 4
      g y z = y * z
```

instead of

```
f 3
WHERE f x = g y z
            WHERE y = ( x + 1 ) * ( x - 1 ) ; z = x ** 2 + 4 ; ;
      g y z = y * z
```

The use of the offside rule poses some constraints on the specification of the unparsing rules by the user of the programming environment. Because of the offside rule the user must be careful when he transforms the layout of some syntactical construct. Some transformations may move offside definitions to non-offside definitions or vice versa.

We restrict our discussion to the specification of the offside rule for the rule for `condexp`. The rule without the offside information is:

```
condexp:
  opexp,
    "->", layout,
      condexp,
        ";", layout,
          condexp.
condexp:
  listexp.
```

In order to specify the offside rule it will be necessary to know a column position. A primitive predicate `column` which returns the current column value during parsing, is therefore available to the specification writer. The initial column value is determined at the beginning of the "->"-symbol in the first alternative of `condexp`. The delimiter, the ";"-symbol in the grammar above, may now be either a ";"-character or a newline character.

```
condexp:
  opexp,
    column (col),
      "->", layout,
        condexp,
          delimiter (col),
            condexp.
```

where

```
delimiter (col):
  ";", layout.
delimiter (col):
  offside (col).
```

The predicate `offside` is defined as:

```
offside (>border):
  column (col),
    smaller (col, border).
```

The definition of the predicate `smaller` can be found in Appendix D.

This approach, forces the layout of the syntactical construct `condexp` into a fixed format:

```
<|opexp|> -> <|condexp|> <|delimiter|>
<|condexp|>
```

The layout will be the same for both orientations, recall Section 4.3.2. Otherwise a program fragment in the vertical orientation

```
<|opexp|> -> <|condexp|> <|condexp|>
```

cannot be recognized when the second alternative for `delimiter` is chosen, because the second `<|condexp|>` is *not* offside. This problem can only be solved if the delimiter is explicitly specified in the rule for `condexp`, one alternative for the delimiter `";"` and one for the implicit delimiter.

```
condexp:
  opexp,
    column (col),
      "->", layout,
        condexp,
          inside (col),
            ";", layout,
              condexp.
```

```
condexp:
  opexp,
    column (col),
      "->", layout,
        condexp,
          delimiter (col),
            condexp.
```

where

```
delimiter (col):
  offside (col).
```

The layout orientation can now be specified for each alternative.

7.4 SASL expressions

The second interesting aspect of the SASL language is the specification of its expressions. Given the BNF specification of SASL the expressions could be specified as:

```
opexp:
  prefix,
    opexp.
opexp:
  opexp,
    infix,
      opexp.
opexp:
  comb.
```

This specification is too simple. The infix and prefix operators do not have the same priorities (Section 7.1) and this rule should therefore be refined, but before we can refine this rule we have to look at the associativity of the operators. Some of the operators will be right associative while others will be left associative. The difference in associativity must be visible in the resulting tree-graph.

The affix position of the first alternative of the hyper rule for `opexp` must be critical to ensure termination of the parsing process. Furthermore, the affix non-terminal must be defined. The part of the eag in which expressions are defined is given below.

```
prio ::  1;2;3;4;5;6;7;8;9.

rightprio ::  1;9.

leftprio ::  2;3;5;6;8.
```

```
preprio ::  4;7.

rightopexp (rightprio):
  opexp (rightprio + 1).
rightopexp (rightprio):
  opexp (rightprio + 1),
    rightinfix (rightprio),
      rightopexp (rightprio).

opexp (>prio):
  opexp (prio + 1).
opexp (rightprio):
  opexp (rightprio + 1),
    rightinfix (rightprio),
      rightopexp (rightprio).
opexp (preprio):
  prefix (preprio),
    opexp (1).
opexp (leftprio):
  opexp (leftprio),
    infix (leftprio),
      opexp (leftprio + 1).
opexp (10):
  comb.
```

The associativity is not the same for all operators. The operators `":"`, `"++"`, `"**"`, and `"."` are right associative, the rest are either left associative or prefix operators. The priorities of the right associative operators are 1 and 9, that of the prefix operators 4 and 7.

The tree-graph for a left associative expression has a different structure than for a right associative expression. We therefore have two different hyper rules. The fact that priorities of the right associative operators are different from those of the left associative operators is used to distinguish between the two rules.

We have left out all other affixes involved identification and specification of operators. This can be found in Appendix D.

7.5 Remaining specification problems

There is one nasty problem in the SASL syntax, viz. an ambiguity at the lexical level which is hard to solve even with affix-directed parsing. It could only be solved by full type checking. Because no type information is available the infix operator `++` can be recognized in two ways, either as the right associative operator `++`, or as the left associative infix operator `+` followed by the prefix operator `+`. In the SASL manual it is stated that the operands of the `++` operator may only be lists, otherwise the result will be `undefined` which is a legal SASL object, see Section 7.2. This problem can be solved in several ways. We have selected the most simple rather than the most elegant one: eliminating the prefix operator `+` from in the SASL language.

The specification of the identification of identifiers is quite straightforward. Identifiers may be multiply defined. A number of identifiers are predefined, such as `abs`, `concat`, `list`, `zip`, etc. The definitions following the `WHERE` add a number of local definitions to the previously defined identifiers.

```
expr (prevdefs):
  expr (locals*prevdefs),
    "WHERE", layout,
      defs (prevdefs, nil, locals).
```

The identifiers in a `namelist` are locally defined with respect to the `expr` following the `"="`-symbol and the surrounding definitions. The same holds for the `name`. However, the identifiers represented by the non-terminal `formal` are only local with respect to the `expr` in the second alternative of `rhs`. This can be specified in the following way:

```
clause (col, prevdefs, locals, newlocals):
  namelist (prevdefs, locals, newlocals),
    column (col),
      "=", layout,
        expr (newlocals*prevdefs).
clause (col, prevdefs, locals, newlocals):
  name (id),
    enter name (id, prevdefs, locals, newlocals),
      rhs (col, prevdefs, newlocals).

rhs (col, prevdefs, locals):
  formal (prevdefs, locals, newlocals),
    rhs (col, prevdefs, newlocals).
rhs (col, prevdefs, locals):
  formal (prevdefs, locals, newlocals),
    column (col),
      "=", layout,
        expr (newlocals*prevdefs).
```

Chapter 8

Conclusions and future work

As we have already stated several times, the emphasis of this research has been on generality and not on efficiency. However, in order to obtain a system which can be used in real life applications the efficiency of the current prototype implementation has to be improved considerably. But before giving the list of improvements and thus hints for future research we draw some conclusions from the preceding chapters.

- Editing programs of ambiguous languages proved not to be feasible in incremental programming environments. In Section 5.3 we gave a number of reasons why ambiguity and incrementality lead to conflicts — at least in the way we wanted to tackle this problem and to assure a consistent tree-graph.
- Deriving a complete programming environment using a simple specification formalism proved to be feasible. In Chapter 4 we gave a number of 'transformation' rules for deriving language specific elements of the environment from the EAG specification.
- The incremental behaviour of the environments generated by PREGMATIC can only be fully exploited when the user has the discipline of editing only small scale constructs via the text edit mode. This will improve performance and it stresses the syntax-directedness of the system.

An advantage of PREGMATIC compared with other systems is its simple specification formalism. This makes PREGMATIC extremely suitable for rapid prototyping, the specification writer is not forced to extend or adapt its specification.

8.1 Generated system

The PREGMATIC-system of course has a number of shortcomings. Some of them can be repaired in a 'final' implementation, others are inherent in the techniques on which the complete concept of the system is based, such as backtracking. The flexibility and functionality of the system and of the user-interface can be improved. These are in fact implementation issues. The first group of shortcomings has to do with efficiency of the resulting system.

- Backtracking should be restricted. The underlying context-free grammar of the eag has to be unambiguous. But the recognition of untyped placeholders and local ambiguous syntactical constructs makes backtracking inevitable. Another parsing technique, or the use of lookahead may also lead to a more efficient result.

- The affix value propagation mechanism is also based on backtracking, but this should be restricted as well. A unification approach in the evaluation mechanism could be considered. However, the full integration of parsing and propagating affix values should be maintained.

8.2 Formalism

In Chapter 1 we stated that we want to use a *given* EAG to generate a programming environment and to safeguard the EAG writer from adapting the specification. This goal has been achieved, except for the specification of the dynamic semantics. Some improvements can still be considered and will be described in the rest of this section.

The EAG-formalism proved to be very suitable as a specification formalism for generating programming environments. The integration of syntax and type checking, and the absence of various environment specific properties makes the formalism concise and easy to read. We have given three methods for specifying the dynamic semantics of a language. To make the interpreter workable it is necessary for the specification writer to mark either the affix positions or the predicates used for the dynamic semantics.

The EAG-formalism nevertheless has a few shortcomings. It is rather inflexible with respect to the tree-graph construction especially if it is compared with the Synthesizer Generator [RT89a]. This system allows the specification writer to manipulate the abstract syntax tree explicitly, which enables him to define, for example, transformation rules within the SSL-formalism. This could be solved by extending the EAG-formalism with facilities for influencing the construction of the tree-graph.

Another shortcoming of the EAG-formalism is the absence of global affixes. This makes it necessary to 'drag' affixes all the way through the specification. In Appendix D this is the case for the affix `prevdefs` in the non-terminal `condexp` and all related non-terminals. The introduction of global affixes would solve this problem of writing 'unnecessary' affix occurrences in the specification.

The unparsing method works satisfactorily for Algol-like languages, which are mainly based on opening and closing keywords. However, for a language such as SASL the user of the generated environment has to adapt the layout of a considerable number of syntactical constructs. This can be prevented by either extending the number of heuristic rules or introducing a mechanism for specifying the unparsing rules in the EAG-formalism.

8.3 Generator

The generation process also has some shortcomings, the efficiency of this process can be considerably improved. The current implementation of the generators is based on a variant of the EAG-formalism, which makes it very suited for the process of prototyping. A re-implementation of these generators in a language like C would make them faster and thus more suited for programming environment generation.

Appendix A

An eag for the EAG-formalism

```
extended affix grammar:
  layout, rules (nil, env).

rules (old env, new env):
  rule (old env, env),
    rules (env, new env).
rules (env, env):  .

rule (old env, new env):
  hyper rule (old env, new env).
rule (env, env):
  affix rule.

hyper rule (old env, new env):
  hyper nonterminal (old env, env),
    ":", layout,
      hyper alternative (env, new env),
        ".", layout.

hyper alternative (old env, new env):
  hyper members (old env, new env).
hyper alternative (env, env):  .

hyper members (old env, new env):
  hyper member (old env, env),
    ",", layout,
      hyper members (env, new env).
hyper members (old env, new env):
  hyper member (old env, new env).

hyper member (old env, new env):
  hyper nonterminal (old env, new env).
hyper member (env, env):
  hyper set.
hyper member (env, env):
  terminal.
```

```
hyper nonterminal (old env, new env):
  nonterminal (identifier),
    display (number),
      consistent (identifier*number, old env, new env).

consistent (>pair, >env, env):
  include (pair, env).
consistent (>pair, >env, pair*env):
  exclude (pair, env).

include (>pair, >pair*env):  .
include (>pair, >head*env):
  differs (pair, head),
    include (pair, env).

exclude (>pair, >nil):  .
exclude (>pair, >head*env):
  differs (pair, head),
    exclude (pair, env).

differs (>new idf*x, >idf*y):
  not equal (new idf, idf).

display (number):
  "(", layout,
    directed affix expressions (number),
      ")", layout.
display (0):  .

directed affix expressions (1 + number):
  directed affix expression,
    ",", layout,
      directed affix expressions (number).
directed affix expressions (1):
  directed affix expression.

directed affix expression:
  ">", layout,
    affix terms.
directed affix expression:
  affix terms.

affix rule:
  affix nonterminal,
    "::", layout,
      affix alternatives,
        ".", layout.

affix alternatives:
  affix expression,
    ";", layout,
      affix alternatives.
affix alternatives:
  affix expression.

affix expression:  .
affix expression:
  affix terms.

affix terms:
  affix term.
affix terms:
  affix term,
    "+", layout,
      affix terms.
affix terms:
  affix term,
    "*", layout,
      affix terms.

affix term:
  affix nonterminal.
affix term:
  affix terminal.
affix term:
  affix set.
affix term:
  affix number.

nonterminal (idf):
  identifier (idf), layout.
```

```
terminal:
  quoted string, layout.

affix nonterminal:
  identifier (idf), layout.

affix terminal:
  quoted string, layout.

hyper set:
  bracket string, layout,
    options,
      display (1).
```

```
affix set:
  bracket string, layout,
    options.

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

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

letgits (ls+lgs):
  { }*!  (blanks),
    {abcdefghijklmnopqrstuvwxyz1234567890}+!  (ls),
      letgits (lgs).
letgits (empty):  .
```

```
quoted string:
  {"} (q1),
    chars,
      {"} (q2).
```

```
bracket string:
  {\{} (b1),
    chars,
      {\}} (b2).
```

```
chars:
  {abcdefghijklmnopqrstuvwxyz}+!  (c),
    chars.
chars:
  {ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789}+!  (c),
    chars.
chars:
  {‘~^_!@#$\{%&*()-+=|[];:’,.<>/?"\} }+!  (c),
    chars.

affix number:
  {123456789} (d),
    {0123456789} (rd),
      layout.

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

Appendix B

The generated parser

```
sym_program()
{
  push_q("program");
  push_q(<<get_program()>>);
  call_q();
  pop_q(2);
}

sym_units()
{
  push_q("units");
  push_q(<<get_units()>>);
  call_q();
  pop_q(2);
}

sym_unit()
{
  push_q("unit");
  push_q(<<get_unit()>>);
  call_q();
  pop_q(2);
}

sym_applicationmarker()
{
  push_q("applicationmarker");
  push_q(<<get_applicationmarker()>>);
  call_q();
  pop_q(2);
  push_q(<<emp_applicationmarker()>>);
  call_q();
  pop_q(1);
}

sym_identifier()
{
  push_q("identifier");
  push_q(<<get_identifier()>>);
  call_q();
  pop_q(2);
}

get_program()
{
  push_q(<<red_program()>>);
  push_q(<<sym_symbol("END")>>);
  push_q(<<sym_units()>>);
  push_q(<<sym_symbol("BEGIN")>>);
  call_q();
  pop_q(4);
}

get_units()
{
  push_q(<<red_unit()>>);
  push_q(<<sym_identifier()>>);
  push_q(<<emp_applicationmarker()>>);
  call_q();
  pop_q(3);
  push_q(<<red_applicationmarker()>>);
  push_q(<<sym_symbol("DEFINE")>>);
  call_q();
  pop_q(2);
  push_q(<<red_applicationmarker()>>);
  push_q(<<sym_symbol("APPLY")>>);
  call_q();
  pop_q(2);
}
```

```
get_applicationmarker()
{
  push_q(<<red_applicationmarker()>>);
  push_q(<<sym_symbol("DEFINE")>>);
  call_q();
  pop_q(2);
  push_q(<<red_applicationmarker()>>);
  push_q(<<sym_symbol("APPLY")>>);
  call_q();
  pop_q(2);
}

get_unit()
{
  push_q(<<red_unit()>>);
  push_q(<<sym_identifier()>>);
  push_q(<<emp_applicationmarker()>>);
  call_q();
  pop_q(3);
  push_q(<<red_applicationmarker()>>);
  push_q(<<sym_symbol("DEFINE")>>);
  call_q();
  pop_q(2);
  push_q(<<red_applicationmarker()>>);
  push_q(<<sym_symbol("APPLY")>>);
  call_q();
  pop_q(2);
}

emp_applicationmarker()
{
  call_q();
}

red_program()
{
  if (top_q() = "program") {
    pop_q(1);
    call_q();
    push_q("program");
  }
}

red_units()
{
  if (top_q() = "units") {
    pop_q(1);
    call_q();
    push_q("units");
  };
  push_q(<<red_units()>>);
  push_q(<<sym_unit()>>);
  push_q(<<sym_symbol(";")>>);
  call_q();
  pop_q(3);
}

red_unit()
{
  if (top_q() = "unit") {
    pop_q(1);
    call_q();
    push_q("unit");
  };
  push_q(<<red_units()>>);
  call_q();
  pop_q(1);
}

red_applicationmarker()
{
  if (top_q() = "applicationmarker") {
    pop_q(1);
    call_q();
    push_q("applicationmarker");
  };
  push_q(<<red_unit()>>);
  push_q(<<sym_identifier()>>);
  call_q();
  pop_q(2);
}

red_identifier()
{
  if (top_q() = "identifier") {
    pop_q(1);
    call_q();
    push_q("identifier");
  }
}
```

Appendix C

An eag for PICO

```
picoprogram:
  layout,
    "BEGIN", layout,
      declarations (decls),
        series (decls),
          "END", layout.
```

```
declarations (decls):
  "DECLARE", layout,
    identifierlist (decls),
      ";", layout.
```

```
identifierlist (decls):
  type (type),
    identifier (id),
      enter declaration (id, type, nil, decls).
identifierlist (new decls):
  identifierlist (old decls),
    ",", layout,
      type (type),
        identifier (id),
          enter declaration (id, type, old decls, new decls).
```

```
series (decls):
  statement (decls).
series (decls):
  series (decls),
    ";", layout,
      statement (decls).

statement (decls):
  assignation (decls).
statement (decls):
  ifstatement (decls).
statement (decls):
  whilestatement (decls).
```

```
assignation (decls):
  identifier (id),
    check application (id, decls, type),
      ":=", layout,
        expression (decls, type).

ifstatement (decls):
  "IF", layout,
    expression (decls, "bool"),
      "THEN", layout,
        series (decls),
          "ELSE", layout,
            series (decls),
              "FI", layout.
```

```
whilestatement (decls):
  "WHILE", layout,
    expression (decls, "bool"),
      "DO", layout,
        series (decls),
          "OD", layout.

expression (decls, type):
  term (decls, type).
expression (decls, "int"):
  expression (decls, "int"),
    "+", layout,
      term (decls, "int").

term (decls, type):
  factor (decls, type).
term (decls, "int"):
  term (decls, "int"),
    "*", layout,
      factor (decls, "int").
```

```
factor (decls, type):
  "(", layout,
    expression (decls, type),
      ")", layout.
factor (decls, type):
  identifier (id),
    check application (id, decls, type).
factor (decls, "int"):
  number.
factor (decls, "bool"):
  boolean.

type ("bool"):
  "BOOL", layout.
type ("int"):
  "INT", layout.
```

```
enter declaration (>id, >type, >decls, newdecls):
  excludes (id, decls),
    add to (decls, id, type, newdecls).

add to (>list, >id, >type, id*type*list):  .

check application (>id, >decls, type):
  includes (id, decls, type).

excludes (>id, >nil):  .
excludes (>id, >head*type*tail):
  not equal (id, head),
    excludes (id, tail).

includes (>id, >id*type*tail, type):  .
includes (>id, >head*htype*tail, type):
  not equal (id, head),
    includes (id, tail, type).

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

```
number:
  {0} ("0"), layout.
number:
  {123456789} (d),
    {0123456789}*!  (ds), layout.

boolean:
  "TRUE", layout.
boolean:
  "FALSE", layout.

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

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

Appendix D

An eag for SASL

```
sasl:
  layout,
    predefines (predefs),
      column (col),
        expr (col, predefs*nil).

expr (margin, prevdefs):
  expr (margin, locals*prevdefs),
    "WHERE", layout,
      defs (prevdefs, nil, locals).
expr (margin, prevdefs):
  condexp (margin, prevdefs).

condexp (margin, prevdefs):
  opexp (1, margin, prevdefs),
    defines (col),
      condexp (prevdefs),
        column (lcol),
          inside (lcol, col),
            ";", layout,
              condexp (margin, prevdefs).
condexp (margin, prevdefs):
  opexp (1, margin, prevdefs),
    defines (col),
      condexp (prevdefs),
        column (lcol),
          offside (lcol, col),
            condexp (margin, prevdefs).

rightopexp (rightprio, margin, prevdefs):
  opexp (rightprio + 1, margin, prevdefs).
rightopexp (rightprio, margin, prevdefs):
  opexp (rightprio + 1, margin, prevdefs),
    rightinfix (rightprio),
      rightopexp (rightprio, margin, prevdefs).

condexp (margin, prevdefs):
  listexp (margin, prevdefs).

listexp (margin, prevdefs):
  opexp (1, margin, prevdefs),
    ",", layout,
      listexp (margin, prevdefs).

defines (col):
  column (col),
    "->", layout.

listexp (margin, prevdefs):
  opexp (1, margin, prevdefs),
    ",", layout.
listexp (margin, prevdefs):
  opexp (1, margin, prevdefs).

prio ::  1;2;3;4;5;6;7;8;9.

rightprio ::  1;9.

leftprio ::  2;3;5;6;8.

preprio ::  4;7.
```

```
opexp (>prio, margin, prevdefs):
  opexp (prio + 1, margin, prevdefs).
opexp (rightprio, margin, prevdefs):
  opexp (rightprio + 1, margin, prevdefs),
    rightinfix (rightprio),
      rightopexp (rightprio, margin, prevdefs).
opexp (preprio, margin, prevdefs):
  prefix (preprio),
    opexp (1, margin, prevdefs).
opexp (leftprio, margin, prevdefs):
  opexp (leftprio, margin, prevdefs),
    infix (leftprio),
      opexp (leftprio + 1, margin, prevdefs).
opexp (10, margin, prevdefs):
  comb (margin, prevdefs).
```

```
prefix (4):
  "~", layout.
prefix (7):
  "-", layout.

rightinfix (1):
  "++", layout.
rightinfix (1):
  ":", layout.
rightinfix (9):
  "**", layout.
rightinfix (9):
  ".", layout.

infix (2):
  "|", layout.
infix (3):
  "&", layout.
infix (5):
  ">>", layout.
infix (5):
  ">", layout.
infix (5):
  ">=", layout.
infix (5):
  "=", layout.
infix (5):
  "~=", layout.
infix (5):
  "<=", layout.
infix (5):
  "<", layout.
infix (5):
  "<<", layout.
```

```
infix (6):
  "+", layout.
infix (6):
  "-", layout.
infix (8):
  "*", layout.
infix (8):
  "/", layout.
infix (8):
  "DIV", layout.
infix (8):
  "REM", layout.

comb (margin, prevdefs):
  comb (margin, prevdefs),
    inside simple (margin, prevdefs).
comb (margin, prevdefs):
  simple (margin, prevdefs).

simple (margin, prevdefs):
  column (lcol),
    inside (lcol, margin),
      inside simple (margin, prevdefs).

inside simple (margin, prevdefs):
  name (id),
    includes in defs (id, prevdefs).
inside simple (margin, prevdefs):
  constant.
inside simple (margin, prevdefs):
  "(", layout,
    expr (margin, prevdefs),
      ")", layout.
```

```
defs (prevdefs, oldlocals, newlocals):
  clause (col, prevdefs, oldlocals, locals),
    column (lcol),
      inside (lcol, col),
        ";", layout,
          defs (prevdefs, locals, newlocals).
defs (prevdefs, oldlocals, newlocals):
  clause (col, prevdefs, oldlocals, locals),
    column (lcol),
      offside (lcol, col),
        defs (prevdefs, locals, newlocals).
defs (prevdefs, oldlocals, newlocals):
  clause (col, prevdefs, oldlocals, newlocals).

clause (col, prevdefs, locals, newlocals):
  namelist (prevdefs, locals, newlocals),
    column (col),
      "=", layout,
        expr (col, newlocals*prevdefs).
clause (col, prevdefs, locals, newlocals):
  name (id),
    enter name (id, prevdefs, locals, newlocals),
      rhs (col, prevdefs, newlocals).

rhs (col, prevdefs, locals):
  formal (prevdefs, locals, newlocals),
    rhs (col, prevdefs, newlocals).
rhs (col, prevdefs, locals):
  formal (prevdefs, locals, newlocals),
    column (col),
      "=", layout,
        expr (col, newlocals*prevdefs).

namelist (prevdefs, oldlocals, newlocals):
  struct (prevdefs, oldlocals, locals),
    ",", layout,
      namelist (prevdefs, locals, newlocals).
namelist (prevdefs, locals, newlocals):
  struct (prevdefs, locals, newlocals),
    ",", layout.
namelist (prevdefs, locals, newlocals):
  struct (prevdefs, locals, newlocals).

struct (prevdefs, oldlocals, newlocals):
  formal (prevdefs, oldlocals, locals),
    ":", layout,
      struct (prevdefs, locals, newlocals).
struct (prevdefs, locals, newlocals):
  formal (prevdefs, locals, newlocals).
```

```
formal (prevdefs, locals, newlocals):
  name (id),
    enter name (id, prevdefs, locals, newlocals).
formal (prevdefs, locals, locals):
  constant.
formal (prevdefs, locals, newlocals):
  "(", layout,
    namelist (prevdefs, locals, newlocals),
      ")", layout.
```

```
string:
  {‘} (q1),
    chars,
      {"} (q2), layout.
```

```
charconst:
  {%} (q),
    char, layout.
```

```
char:
  {abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789} (c).
char:
  {‘~^_!@#$%&\{*()-+=|[];:’,.<>/?"\} } (c).

chars:  .
chars:
  {abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789}+!  (c),
    chars.
chars:
  {‘~^_!@#$%&\{*()-+=|[];:’,.<>/?\} }+!  (c), chars.
```

```
constant:
  numeral.
constant:
  charconst.
constant:
  boolconst.
constant:
  "()", layout.
constant:
  string.

numeral:
  real,
    scalefactor, layout.
numeral:
  "-", layout,
    real,
      scalefactor, layout.
```

```
real:
  {0123456789}+!  (ds).
real:
  {0123456789}+!  (ds),
    {.} (p),
      {0123456789}+!  (fds).

scalefactor:  .
scalefactor:
  {e} (e),
    {-} (s),
      {0123456789}+!  (ds).
scalefactor:
  {e} (e),
    {0123456789}+!  (ds).
```

```
boolconst:
  "TRUE", layout.
boolconst:
  "FALSE", layout.

name (l+ls):
  {abcdefghijklmnopqrstuvwxyz} (l),
    {abcdefghijklmnopqrstuvwxyz0123456789_}*!  (ls), layout.

layout:
  { \n}*!  (ignored).

offside (>col, >col+1+x):  .

inside (>col+x, >col):  .

enter name (>id, >prevdefs, >locals, locals):
  includes in defs (id, prevdefs).
enter name (>id, >prevdefs, >locals, newlocals):
  excludes in defs (id, prevdefs),
    enter local name (id, locals, newlocals).

enter local name (>id, >locals, locals):
  includes (id, locals).
enter local name (>id, >locals, newlocals):
  excludes (id, locals),
    addto (id, locals, newlocals).

includes (>id, >id*rest):  .
includes (>id, >head*tail):
  not equal (id, head),
    includes (id, tail).

excludes (>id, >nil):  .
excludes (>id, >head*tail):
  not equal (id, head),
    excludes (id, tail).

includes in defs (>id, >locals*restdefs):
  includes (id, locals).
includes in defs (>id, >locals*restdefs):
  excludes (id, locals),
    includes in defs (id, restdefs).

excludes in defs (>id, >nil):  .
excludes in defs (>id, >locals*restdefs):
  excludes (id, locals),
    excludes in defs (id, restdefs).

addto (>id, >list, id*list):  .
```

```
predefines (defs):
  addto ("abs", nil, def1),
  addto ("all", def1, def2),
  addto ("and", def2, def3),
  addto ("append", def3, def4),
  addto ("arctan", def4, def5),
  addto ("code", def5, def6),
  addto ("cons", def6, def7),
  addto ("converse", def7, def8),
  addto ("char", def8, def9),
  addto ("cjustify", def9, def10),
  addto ("concat", def10, def11),
  addto ("cos", def11, def12),
  addto ("count", def12, def13),
  addto ("decode", def13, def14),
  addto ("digit", def14, def15),
  addto ("digitval", def15, def16),
  addto ("drop", def16, def17),
  addto ("e", def17, def18),
  addto ("entier", def18, def19),
  addto ("eq", def19, def20),
  addto ("exp", def20, def21),
  addto ("filter", def21, def22),
  addto ("foldl", def22, def23),
  addto ("foldr", def23, def24),
  addto ("for", def24, def25),
  addto ("from", def25, def25),
  addto ("function", def25, def26),
  addto ("hd", def26, def27),
  addto ("I", def27, def28),
  addto ("interleave", def28, def29),
  addto ("intersection", def29, def30),
  addto ("iterate", def30, def31),
  addto ("K", def31, def32),
  addto ("lay", def32, def33),
  addto ("layn", def33, def34),
  addto ("length", def34, def35),
  addto ("letter", def35, def36),
  addto ("list", def36, def37),
  addto ("ljustify", def37, def38),
  addto ("log", def38, def39),
  addto ("logical", def39, def40),
  addto ("member", def40, def41),
  addto ("modulo", def41, def42),
  addto ("not", def42, def43),
  addto ("number", def43, def44),
  addto ("or", def44, def45),
  addto ("pi", def45, def46),
  addto ("plus", def46, def47),
  addto ("printwidth", def47, def48),
  addto ("product", def48, def49),
  addto ("reverse", def49, def50),
  addto ("show", def50, def51),
  addto ("sin", def51, def52),
  addto ("some", def52, def53),
  addto ("spaces", def53, def54),
  addto ("sqrt", def54, def55),
  addto ("sum", def55, def56),
  addto ("take", def56, def57),
  addto ("times", def57, def58),
  addto ("tl", def58, def59),
  addto ("union", def59, def60),
  addto ("until", def60, def61),
  addto ("while", def61, def62),
  addto ("zip", def62, defs).
```

Bibliography

[Akk89] R. op den Akker. *Parsing Attribute Grammars.* PhD thesis, University of Twente, 1989.

[Alb89a] H. Alblas. Attributed tree transformations with delayed and smart re-evaluation. In D. Hammer, editor, *Compiler Compilers and High Speed Compilation*, volume 371 of *Lecture Notes in Computer Science*, pages 160–174. Springer Verlag, Berlin Heidelberg New York, 1989.

[Alb89b] H. Alblas. Iteration of transformation passes over attributed program trees. *Acta Informatica*, 27:1–40, 1989.

[Alb89c] H. Alblas. Optimal incremental simple multi-pass attribute evaluation. *Information Processing Letters*, 32:289–295, 1989.

[Alb91a] H. Alblas. Incremental attribute evaluation. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 215–233. Springer Verlag, Berlin Heidelberg New York, 1991.

[Alb91b] H. Alblas. Introduction to attribute grammars. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 1–15. Springer Verlag, Berlin Heidelberg New York, 1991.

[AMT91] R. op den Akker, B. Melichar, and J. Tarhio. Attribute Evaluation and Parsing. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 187–214. Springer Verlag, Berlin Heidelberg New York, 1991.

[AU72] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation, and Compiling, Volume I: Parsing.* Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[BCD+89] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practice Software Development Environments*, pages 14–24, 1989. Appeared as *SIGPLAN Notices* 14(2).

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification*. ACM Press in co-operation with Addison-Wesley, 1989.

[BLM89] M.G.J. van den Brand, J.J. Langeveld, and H. Meijer. Syntactical and static semantical error reporting in automatically generated backtrack parsers. In *Proceedings of the Annual Conference Computing Science in the Netherlands*, pages 57–70, Utrecht, November 1989.

[BMS87] R. Bahlke, B. Moritz, and G. Snelting. A Generator for Language-Specific Debugging Systems. *SIGPLAN Notices*, 22(7):92–101, 1987.

[Bra90] M.G.J. van den Brand. Incremental affix evaluation in syntax directed editors. In *Proceedings of the Annual Conference Computing Science in the Netherlands*, pages 35–51, Utrecht, November 1990.

[BS84] P.A. Bailes and A. Salvadori. A Semantically-based Formatting Discpline for Pascal. *Software—Practice and Experience*, 14(3):235–251, 1984.

[BS85] R. Bahlke and G. Snelting. The PSG – Programming System Generator. Proc. ACM Symposium on Language Issues in Programming Environments. *SIGPLAN Notices*, 20(7):28–33, 1985.

[BS86] R. Bahlke and G. Snelting. The PSG System: From Formal Language Definitions to Interactive Programming Environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.

[BS89] G. Blaschek and J. Sametinger. User-adaptable Prettyprinting. *Software—Practice and Experience*, 19(7):687–702, 1989.

[CK84] R.H. Campbell and P.A. Kirslis. The SAGA Project. *SIGPLAN Notices*, 19(5):73–80, 1984.

[Deu91] A. van Deursen. An algebraic specification for the static semantics of Pascal. Technical Report CS-R9129, CWI, Amsterdam, 1991.

[Hal91] H. van Halteren. Efficient storage of ambiguous structures in textual databases. Technical report, Dept. of Language and Speech, University of Nijmegen, 1991.

[Hee83] J. Heering. Taaldefinities als kern voor een programmeeromgeving. In J. Heering and P. Klint, editors, *Colloquium Programmeeromgevingen*, volume MC Syllabus 30, pages 69–81. CWI, Amsterdam, 1983.

[Hen91] P.R.H. Hendriks. *Implementation of Modular Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

[HH90] H. van Halteren and Th. van den Heuvel. *Linguistic Exploitation of Syntactic Databases. The Use of the Nijmegen Linguistic DataBase Program.* Rodopi, Amsterdam, 1990.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[Hie87] A. Hiemstra. Een compiler voor Miranda in EAG. Master's thesis, Katholieke Universiteit Nijmegen, August 1987. In Dutch.

[HK86] J. Heering and P. Klint. A syntax definition formalism. Technical Report CS-R8633, CWI, Amsterdam, 1986.

[HKKL86] J. Heering, G. Kahn, P. Klint, and B. Lang. Generation of interactive programming environments. In *ESPRIT '85: Status Report of Continuing Work*, pages 467–477. North-Holland, 1986.

[HKR87] J. Heering, P. Klint, and J. Rekers. Incremental generation of lexical scanners. Technical Report CS-R8761, CWI, Amsterdam, 1987.

[HKR90] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, SE-16(12):1344–1351, 1990.

[HKR91] J. Heering, P. Klint, and J. Rekers. Lazy and incremental program generation. Technical Report CS-R9124, CWI, Amsterdam, 1991.

[HN86] A.N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, SE-12:1117–1127, 1986.

[HSV86] J. Heering, J. Sidi, and A. Verhoog. Generation of interactive programming environments - GIPE. Technical Report CS-R8620, CWI, Amsterdam, 1986.

[Joh75] S.C. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[Jok89] M.O. Jokinen. A Language-independent Prettyprinter. *Software—Practice and Experience*, 19(9):839–856, 1989.

[JW86] K. Jensen and N. Wirth. *Pascal User Manual and Report, ISO Pascal Standard.* Springer Verlag, Berlin Heidelberg New York, 1986. Third edition.

[Kai89] G.E. Kaiser. Incremental Dynamic Semantics for Language-Based Programming Environments. *ACM Transactions on Programming Languages and Systems*, 11(2):169–193, 1989.

[Kas91] U. Kastens. Implementation of Visit-Oriented Attribute Evaluators. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute*

Grammars, Applications and Systems, volume 545 of *Lecture Notes in Computer Science*, pages 114–139. Springer Verlag, Berlin Heidelberg New York, 1991.

[KB91] C.H.A. Koster and J.G. Beney. On the borderline between grammars and programs. In *Proceedings of the International Symposium on Programming Language Implementation and logic Programming*, volume 528 of *Lecture Notes in Computer Science*, Passau (FRG), 1991. Springer Verlag, Berlin Heidelberg New York.

[KBHG+87] B. Krieg-Brückner, B. Hoffmann, H. Ganzinger, M. Broy, R. Wilhelm, U. Möncke, B. Weisgerber, A. McGettrick, I.G. Campbell, and G. Winterstein. PROgram development by SPECification and TRAnsformation. In *Proc. ESPRIT Conf. '86 (Results and Achievements)*, pages 301–312. North-Holland Publishing Company, Amsterdam, 1987.

[Kli83] P. Klint. A survey of three language-independent programming environments. Technical Report CS-R83240, CWI, Amsterdam, 1983.

[Kli91] P. Klint. A meta-environment for generating programming environments. In J.A. Bergstra and B. Feijs, editors, *Proceedings of the METEOR workshop on Methods Based on Formal Specification*, volume 490 of *Lecture Notes in Computer Science*, pages 105–124. Springer Verlag, Berlin Heidelberg New York, 1991.

[Knu68] D.E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2:127–145, February 1968. See also [Knu71].

[Knu71] D.E. Knuth. Semantics of Context-Free Languages: Correction. *Mathematical Systems Theory*, 5:95–96, May 1971.

[Kos71] C.H.A. Koster. Affix Grammars. In J.E.L. Peck, editor, *Algol68 Implementation*, pages 95–109. North Holland Publishing Company, Amsterdam, 1971.

[Kos75] C.H.A. Koster. A Technique for Parsing Ambiguous Grammars. In D. Siefkes, editor, *GI—4.Jahrestagung*, volume 26 of *Lecture Notes in Computer Science*, pages 233–246. Springer Verlag, Berlin Heidelberg New York, 1975.

[Kos91a] C.H.A. Koster. Affix Grammars for Natural Languages. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 469–484. Springer Verlag, Berlin Heidelberg New York, 1991.

[Kos91b] C.H.A. Koster. Affix Grammars for Programming Languages. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 358–373. Springer Verlag, Berlin Heidelberg New York, 1991.

[KW86] C.H.A. Koster and Th.P. van der Weide. A programming environment for secondary schools. Technical Report 99, Katholieke Universiteit Nijmegen, 1986.

[LDHH78] J. Lewi, K. De Vlaminck, J. Huens, and M. Huybrechts. The ELL(1) Parser Generator and the Error Recovery Mechanism. *Acta Informatica*, 10:209–228, 1978.

[Lea84] G.T. Leavens. Prettyprinting Styles for Various Languages. *SIGPLAN Notices*, 19(2):75–79, 1984.

[Les75] M.E. Lesk. Lex - A Lexical Analyzer Generator. Technical Report Computer Science No. 39, Bell Laboratories, Murray Hill, New Jersey, 1975.

[LMOW88] P. Lipps, U. Möncke, M. Olk, and R. Wilhelm. Attribute (Re)evaluation in OPTRAN. *Acta Informatica*, 26:213–239, 1988.

[Log88] M. Logger. An integrated text and syntax-directed editor. Technical Report CS-R8820, CWI, Amsterdam, 1988.

[Mat83] P. Mateti. A Specification Schema for Indenting Programs. *Software—Practice and Experience*, 13:163–179, 1983.

[Med82] R. Medina-Mora. *Syntax directed editing: Towards integrated programming environments*. PhD thesis, Carnegie-Mellon University, 1982.

[Mei86] H. Meijer. *Programmar: A Translator Generator*. PhD thesis, Katholieke Universiteit Nijmegen, 1986.

[Mei90] H. Meijer. The project on EXTENDED AFFIX GRAMMARS at Nijmegen. In P. Deransart and M. Jourdan, editors, *Proceedings of International Conference WAGA on Attribute Grammars and their Applications*, volume 461 of *Lecture Notes in Computer Science*, pages 130–142. Springer Verlag, Berlin Heidelberg New York, 1990.

[Mei92] H.J.M. Meijer. *Calculating Compilers*. PhD thesis, Katholieke Universiteit Nijmegen, 1992.

[Meu90] E.A. van der Meulen. Deriving incremental implementations from algebraic specifications. Technical Report CS-R9072, CWI, Amsterdam, 1990.

[MF81] R. Medina-Mora and P. Feiler. An incremental programming environment. *IEEE Transactions on Software Engineering*, 7(5):472–482, 1981.

[MK62] L.G.L.Th. Meertens and C.H.A. Koster. An affix grammar for a part of the English language. presented at Euratom Colloquium, University of Amsterdam, 1962.

[Mor89] M.P.G. Moritz. *Description and Analysis of Static Semantics by Fixed Point Equations*. PhD thesis, Katholieke Universiteit Nijmegen, 1989.

[Ned91] M.-J. Nederhof. Left-Corner Parsing Rehabilitated? 1991. submitted.

[NS91] M.-J. Nederhof and J. Sarbo. Partial Evaluation Grammars. Technical Report 91-1, Katholieke Universiteit Nijmegen, 1991.

[Opp80] D.C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, 1980.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[Rep84] T.W. Reps. *Generating Language-Based Environments*. MIT Press, 1984.

[RT89a] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator: a system for constructing language-based editors*. Springer Verlag, Berlin Heidelberg New York, 1989.

[RT89b] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual*. Springer Verlag, Berlin Heidelberg New York, 1989. Third edition.

[RTD83] T.W. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, 1983.

[Rub83] L.F. Rubin. Syntax-directed pretty printing—a first step towards a syntax directed editor. *IEEE Transactions on Software Engineering*, SE-9:111–127, 1983.

[RW81] G.A. Rose and J. Welsh. Formatted Programming Languages. *Software—Practice and Experience*, 11:651–669, 1981.

[Sun90a] Sun Microsystems, Inc./AT&T. *OPEN LOOK Graphical User Interface, Application Style Guidelines*. Addison-Wesley, Reading, Massachusetts, 1990.

[Sun90b] Sun Microsystems, Inc./AT&T. *OPEN LOOK Graphical User Interface, Functional Specification*. Addison-Wesley, Reading, Massachusetts, 1990.

[Tip91] F. Tip. The equation debugger. Master's thesis, University of Amsterdam, 1991.

[TR81] T. Teitelbaum and T.W. Reps. The Cornell Program Synthesizer: syntax directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[Tur79] D.A. Turner. SASL Language Manual. Technical report, University of Kent, Canterbury, 1979.

[Tur90] D.A. Turner. An Overview of Miranda. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 1–16. Addison-Wesley, Reading, Massachusetts, 1990.

[VBF90] H. Vogt, A. van den Berg, and A. Freije. Rapid development of a program transformation system with attribute grammars and dynamic transformations. In P. Deransart and M. Jourdan, editors, *Proceedings of International Conference WAGA on Attribute Grammars and their Applications*, volume 461 of *Lecture Notes in Computer Science*, pages 101–115. Springer Verlag, Berlin Heidelberg New York, 1990.

[VSK89] H.H. Vogt, S.D. Swierstra, and M.F. Kuiper. Higher Order Attribute Grammars. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 131–145, Portland, Oregon, 1989.

[VSK91] H.H. Vogt, S.D. Swierstra, and M.F. Kuiper. Efficient incremental evaluation of higher order attribute grammars. In *Proceedings of the International Symposium on Programming Language Implementation and logic Programming*, Passau (FRG), 1991.

[Wal91] H.R. Walters. *On Equal Terms.* PhD thesis, University of Amsterdam, 1991.

[Wat74] D.A. Watt. *Analysis-Oriented Two-Level Grammars.* PhD thesis, University of Glasgow, January 1974.

[Wat79] D.A. Watt. An extended attribute grammar for Pascal. *SIGPLAN Notices*, 14(2):60–74, 1979.

[WJ88] J.A. Walz and G.F. Johnson. Incremental Evaluation for a General Class of Circular Attribute Grammars. In *ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 209–221, Atlanta, Georgia, 1988.

[WMP+76] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, and R.G. Fisker. *Revised Report on the Algorithmic Language* ALGOL 68. Springer Verlag, Berlin Heidelberg New York, 1976.

[Woo86] M. Woodman. Formatted Syntaxes and Modula-2. *Software—Practice and Experience*, 16(7):605–626, 1986.

[Zwo90] A.J. van Zwol. AGFL's revisited—a parser system with integrated lexicon. Master's thesis, Katholieke Universiteit Nijmegen, 1990.

Index

Samenvatting

PREGMATIC is een generator voor incrementele programmeeromgevingen dat wil zeggen een programma dat door middel van transformaties een formele beschrijving van een willekeurige taal (programmeer- of specificatietaal) in een programmeeromgeving voor deze taal omzet. Het formalisme waarin de taal beschreven wordt, is Extended Affix Grammars (EAGs) [Mei86]. Dit formalisme is ontwikkeld voor de beschrijving van zowel de syntax als de statische en dynamische semantiek van een taal. In vergelijkbare systemen, zoals de Synthesizer Generator [RT89a] en PSG [BS85, BS86], worden formalismen gebruikt die zeer dicht tegen Extended Affix Grammars aan liggen: attribuutgrammatica's. Het specificatieformalisme SSL [RT89b] is een attribuutgrammatica uitgebreid met allerlei faciliteiten om de diverse onderdelen van een programmeeromgeving te beschrijven, zoals de abstracte syntax, gaten (placeholders) en sjablonen (templates). Het gevolg van deze aanpak is dan ook niet meer het abstracte beschrijven van een taal maar concreet uitprogrammeren van een omgeving. Een andere techniek om talen te beschrijven voor het genereren van programmeeromgevingen gaat uit van algebraische specificaties. Het specificatieformalisme ASF+SDF [HHKR89, Hen91] (ontwikkeld binnen het GIPE-project) is hier een voorbeeld van.

Een van de doelstellingen van het onderzoek beschreven in dit proefschrift was het EAG-formalisme onaangetast te laten, dat wil zeggen niet uit te breiden met allerlei toeters en bellen om de diverse programmeeromgeving-afhankelijke faciliteiten uit te programmeren.

In Hoofdstuk 1 hebben we een overzicht gegeven van de diverse andere systemen waarmee programmeeromgevingen kunnen worden gegenereerd De belangrijkste systemen zijn de Synthesizer Generator [RT89a], PSG [BS85, BS86], GANDALF [MF81, Med82] en de ASF+SDF meta-environment [Kli91].

In Hoofdstuk 2 is het EAG-formalisme besproken. De beschrijving van het formalisme bestaat uit een syntactisch en een semantisch gedeelte, waarbij het semantische gedeelte de beschrijving bevat van consistente substitutie en predicaten. Het principe van affix-gestuurd ontleden is geïntroduceerd en er wordt een aantal voorbeelden genoemd waarbij dit principe toegepast kan worden. Affix-gestuurd ontleden wil zeggen dat er tijdens het ontleden van de invoerzin al berekeningen worden gedaan, bijvoorbeeld of er statisch semantische conflicten optreden. Verder wordt een gedetailleerde beschrijving van het typeringsmechanisme van EAGs gegeven.

De Hoofdstukken 3 en 5 bespreken een concreet executiemodel van EAGs dat als basis voor de gegenereerde programmeeromgevingen kan dienen. Dit gebeurt in een aantal stappen uitgaande van een gegeven EAG. Eerst wordt een ontleder (left-corner back-

track ontleder) gegenereerd uit de onderliggende context-vrije grammatica. Daarna wordt beschreven hoe de interne datastructuren (syntaxboom en affixgraaf) eruitzien. Vervolgens wordt een aantal algoritmen beschreven die de knopen in de opgebouwde affixgraaf van affixwaarden voorzien. Zowel de ontleder als de evaluatie-algoritmen zijn op backtracking gebaseerd, beide werken zeer nauw samen om affix-gestuurd te kunnen ontleden. In Hoofdstuk 3 wordt ervan uitgegaan dat na iedere edit-operatie de hele invoerzin opnieuw wordt bekeken. In Hoofdstuk 5 wordt besproken hoe informatie uit eerder uitgevoerde berekeningen gebruikt kan worden, hetgeen resulteert in incrementele versies van de diverse algoritmen. Met name het incrementele evaluatie-algoritme is interessant. Incrementaliteit is een van de belangrijkste eigenschappen van gegenereerde programmeeromgevingen. Na iedere verandering in de programmatekst wordt het ontleden tot een minimum beperkt. Door aan het affix-evaluatie-mechanisme een stap toe te voegen waarin de aangetaste knopen worden gemarkeerd, wordt van een zo klein mogelijk gedeelte van de affixgraafknopen de waarde opnieuw berekent.

In Hoofdstuk 4 wordt de structuur van de gegenereerde omgeving besproken. Behalve naar de taalonafhankelijkheden, zoals het gebruikersinterface, wordt er gekeken naar taalafhankelijkheden zoals de generatie van sjablonen en (getypeerde en ongetypeerde) gaten. De prettyprint wordt automatisch afgeleid uit de onderliggende context-vrije grammatica. Pregmatic biedt echter de mogelijkheid om deze afgeleide prettyprint voor de afzonderlijke taalconstructies te veranderen. Het hele prettyprintmechanisme wordt eveneens beschreven in dit Hoofdstuk.

In Hoofdstuk 6 wordt beschreven welke mogelijkheden het EAG-formalisme biedt voor het executeren van programma's ontwikkeld met een programmeeromgeving gegenereerd met Pregmatic. Er worden drie verschillende manieren besproken om de dynamische semantiek van een taal te beschrijven in EAGs. Geen van deze drie modellen is geïmplementeerd in het prototype, omdat voor ieder van deze modellen een aanpassing van het EAG-formalisme onvermijdelijk zal zijn. Verder wordt er nog een kort overzicht gegeven van vergelijkbare technieken toegepast in andere programmeeromgevingsgeneratoren.

Voor het testen van Pregmatic en de gegenereerde programmeeromgevingen zijn diverse talen, gespecificeerd met behulp van EAGs, gebruikt. Hoofdstuk 7 beschrijft de functionele taal SASL [Tur79]. Niet alleen de syntax en semantiek worden beschreven, maar ook problemen die optraden tijdens het specificeren van SASL in EAGs. In de Appendices zijn nog twee andere EAG-specificaties opgenomen, namelijk de beschrijving van het EAG-formalisme zelf (Appendix A) en de beschrijving van het speelgoedtaaltje PICO (Appendix C).

De belangrijkste doelstelling van het onderzoek was het formalisme eenvoudig te houden en de mogelijkheden te bekijken om uit zo'n eenvoudig formalisme zoveel mogelijk onderdelen van de omgeving te genereren. De efficientie van de gegenereerde programmeeromgevingen en de generatoren is niet aan bod gekomen in dit onderzoek. Het huidige prototype van Pregmatic kan als uitgangspunt dienen om een generator te ontwikkelen die efficientere programmeeromgevingen genereert.

Curriculum Vitæ

27 maart 1962
Geboren te Born (Limburg).

22 juni 1978
MAVO-diploma, R.K. Mavo Pius X te Susteren.

29 mei 1980
HAVO-diploma, "Serviam" R.K. Scholengemeenschap Lyceum-HAVO te Sittard.

4 juni 1982
VWO-diploma, "Serviam" R.K. Scholengemeenschap Lyceum-HAVO te Sittard.

26 juni 1987
Doctoraaldiploma, Informatica, Katholieke Universiteit Nijmegen.

1 juli 1987 — 30 juni 1989
Onderzoeker in opleiding, Informatica, Katholieke Universiteit Nijmegen.

2 augustus 1989 — 31 maart 1992
Junior onderzoeker, Informatica, Katholieke Universiteit Nijmegen.

1 april 1992
Universitair docent, Vakgroep Programmatuur, Universiteit van Amsterdam.