Informal introduction to the Extended Affix Grammar formalism and its compiler

M. Seutter

October 11, 2001

Chapter 1

Introduction

This book intends to introduce the Extended Affix Grammar formalism [Wat74] and its compiler. The Extended Affix Grammar formalism, or shortly EAG, is a formalism for describing both the context free and the context sensitive syntax of languages. EAG is a member of the family of two-level grammars. They are very closely related to two-level van Wijngaarden grammars [Wij76]. To a description in EAG the EAG compiler may be applied, generating either a recognizer or a transducer or a translator or a syntax directed editor for the language described.

We will not discuss the exact details of the formalism, but we will pay more attention to the ideas behind the language and elucidate the language concepts by many examples. All these examples can be found in the example subdirectory (`examples`) of the compiler package. We will also discuss the use of the compiler and the options that may be used to steer the generation process. We will assume that the reader has some basic knowledge of context free grammars.

In the following chapters we will use the `typewriter` font to indicate examples in EAG. In chapter 2 we will discuss context free descriptions in EAG. Chapter 3 will introduce the second level of the formalism. The next chapter introduces meta rules, which may impose a context free grammar on the second level of the formalism. In chapter 5 we will see how the compiler may be used for generating a syntax directed editor from an EAG description. Finally, chapter 6 indicates the direction of future research.

There are 3 appendices. Appendix A gives some more details of the package (It is useful to read the first two sections if you wish to install the package). Appendix B gives a description of the EAG formalism in EAG. Finally the last appendix contains some references.

This is the first version of a new major release (2.0) of the EAG compiler. As such, it is quite experimental and bugs may turn up. Bug reports may therefore be sent to marcs@cs.kun.nl, so that these bugs may be corrected in later versions.

Copyright Notice

Warranty

Because the EAG compiler package is licensed free of charge, we provide absolutely no warranty, to the extent permitted by applicable law. The University of Nijmegen provides the EAG compiler package "as is" without warranty of any kind.

Distribution

The EAG compiler package is available by ftp at `ftp.cs.kun.nl` in the directory `/pub/eag`.

Chapter 2

Describing context free languages

2.1 Basic syntax

An EAG consists of *syntax rules*, *affix production rules* (also called *meta rules)* and an optional *start rule*. In this chapter we will focus on the syntax rules. A syntax rule consists of one or more *syntax rule elements*, separated by semicolons, ending in a period. A syntax rule element consists of a left hand side, a colon and one or more *alternatives*. Alternatives are separated by semicolons. The left hand side of a syntax rule element consists of a *nonterminal* identifying the syntax rule. For the present, nonterminals are identifiers consisting of letters and digits. Blanks may occur in nonterminals but they are ignored. Thus `even more elements` and `evenmoreelements` denote the same name.

An alternative consists of zero or more members, separated by commas. A member can either be a *terminal*, a nonterminal, a *semiterminal* or a *commit operator*. A terminal is a sequence of characters, enclosed between quotes. Semiterminals and the commit operator will be discussed later. When reading an EAG, one can read the colon as "is defined as", the semicolon as "or", the comma as "followed by" and the period as "end of definition".

The reader may perhaps wonder why there are two syntactic ways in EAG to indicate choice namely the semicolon between syntax rule elements and the semicolon between alternatives. The explanation of the reason behind this must unfortunately be postponed to chapter 3. Consider the following example:

```
smengels:
   subject, verb, ".".
```

```
subject:
   article, noun.

article: "the "; "a ".

noun: "man ";
noun: "dog ".

verb: "sleeps"; "sits".
```

As we can see this grammar specifies a language of the following 8 sentences:

- the man sleeps
- a man sleeps
- the dog sleeps
- a dog sleeps
- the man sits
- a man sits
- the dog sits
- a dog sits

EAGs must obey the following context conditions:

- All left hand side identifiers of one syntax rule must be the same.
- The left hand side identifiers of different syntax rules must differ.
- For every nonterminal occurring in an alternative there must be a syntax rule (or possibly a builtin syntax rule) with the same name.

We will call the application of a nonterminal on the right hand side a *call* of that nonterminal. The start rule of an EAG consists of a single call of the starting syntax rule, followed by a period. There are three (evident) context conditions concerning start rules:

- For the nonterminal in the start rule, there must be a (possibly builtin) syntax rule with the same name.
- More than one start rule is not allowed.
- If an EAG does not specify a start rule, the textually first syntax rule of the EAG will be taken as start rule.

2.2 A first use of the compiler

Suppose we have the file `smengels.eag` in the example directory which contains the above syntax rules. We can compile this grammar by entering the following two commands:

```
eag-compile smengels
eagcc smengels_leftcorner
```

The first of these commands compiles the grammar into a recognizer in the programming language C. The second one compiles this recognizer into an executable, called `smengels_leftcorner` (Later we will discuss why this name is given to the executable). If one executes this file, it will read lines from standard input until an end of file is encountered. This input will then be parsed by the recognizer, which will then report whether the parse was succesfull or not.

Unfortunately, lines read from stdin in Unix end in a newline. Thus the recognizer will fail and report that no parse could be found. We therefore adapt the above grammar in the following way (as given in the file `smengels2.eag` in the `examples` subdirectory):

```
smengels2.

smengels2:
   subject, verb, ".", nlcr.

subject:
   article, noun.

article: "the "; "a ".

noun: "man ";
noun: "dog ".

verb: "sleeps"; "sits".
```

In this example we use the builtin syntax rule `nlcr` to specify the trailing newline. If we compile the above example we will see that the recognizer will not produce any error message for grammatically correct input sentences. If the recognizer can not find any correct parse it will generate an error message including the farthest position in the input upto which it could parse.

The EAG compiler will automatically add a check in the start rule to match the end of file at the end of the input. Thus it is impossible to report a successfull parse, if any unconsumed input remains.

The EAG compiler generates *recursive backup* parsers. Such a parser uses recursive procedures for parsing. The parser backtracks when the recognition of a symbol fails.

Backtracking means undoing the effect of one or more calls the parser has lastly done. If recognition of the whole input succeeds, the parser reports a “successful parse” and then backtracks. If at a certain point several steps are possible they will be tried one after the other.

This means that a parser generated by the compiler will find all possible parses of the input. Thus the compiler can handle *ambiguous* grammars (A context free grammar is called ambiguous if there exists a string with more than one distinct derivations according to that grammar). This is one of the great advantages of the EAG compiler since many interesting languages are inherently ambiguous, such as natural languages. The disadvantage of a backtracking parser is its high (exponential) time complexity. However the EAG compiler codes several heuristics, such as lookahead, in the generated parser to limit the amount of backtracking while parsing. An example of an ambiguous grammar is the following (to be found in the file `examples/amb_expr.eag`):

```
start.

start: expr, nlcr.

expr: expr, "+", expr.
expr: term.

term: term, "*", term.
term: id.

id: "a"; "b"; "c"; "d"; "e"; "f".
```

If we enter `a+b+c` as input the generated parser will report two successful parses. If the user wants to inspect the different derivations, she must add the flag `-D` when compiling the grammar:

```
eag-compile -D amb_expr
eagcc amb_expr_leftcorner
```

When parsing the input `a+b+c` the parser will give the two parse trees corresponding to the two derivations:

```
Dump of parse tree:
rootnode (0)
 rule_start (11)
  rule_expr (3)
   rule_expr (3)
    rule_expr (4)
     rule_term (13)
      rule_id (5)
    rule_expr (4)
     rule_term (13)
      rule_id (6)
   rule_expr (4)
    rule_term (13)
     rule_id (7)
  rule_nlcr (2)
```

```
Dump of parse tree:
rootnode (0)
 rule_start (11)
  rule_expr (3)
   rule_expr (4)
    rule_term (13)
     rule_id (5)
   rule_expr (3)
    rule_expr (4)
     rule_term (13)
      rule_id (6)
    rule_expr (4)
     rule_term (13)
      rule_id (7)
  rule_nlcr (2)
```

Such a dump of the parse tree will only show nonterminals (Terminals are not stored in the parse tree built by the parser). The number behind the nonterminal indicates which alternative of a syntax rule is taken (All alternatives in an EAG are given a unique number by the parser generator. This number is shown in the dump).

Of course the grammar writer can make errors from time to time when writing the grammar. Many of these will be flagged by the compiler as syntax or identification errors. Sometimes it is useful to obtain a trace of the execution of a parser. One can obtain a trace by adding the flag `-T` at compilation time:

```
eag-compile -T amb_expr
eagcc amb_expr_leftcorner
```

An example of this trace is the following piece of a trace obtained from a parser for the above grammar for the input `a+b+c`:

```
13 '\n': <red_id
12 'c': <get_expr
11 'c': <rule_expr
10 '+': <red_expr
10 '+': -rule_expr, alt 1
10 'c': >rule_expr
11 'c': >get_expr
12 'c': -rule_id, alt 3
12 '\n': >red_id
```

```
:
:
18 '\n': >get_nlcr
19 '\n': -rule_nlcr, alt 1
19 '\e': >red_nlcr
20 '\e': >red_start
A parse was found
20 '\e': <red_start
```

The leftmost number of each line indicates the depth in recursion, followed by the first unconsumed character in the input. The end of file character is indicated by `\e`, as can be seen in the example. The remainder of the line indicates the parsing action that is being executed at that point of the parse. This example also shows the recursive backup nature of the parser (We can see that the parser backups until the `+` is reached again and then continue parsing according to another alternative).

2.3 Two kinds of parsers

The EAG compiler can generate two kinds of parsers namely

- topdown parsers
- leftcorner parsers

Topdown parsers can be obtained by adding the flag `-td` to the command line when generating the parser. The kind of the parser is added to the name of the grammar when generating the parser. Suppose we want to generate a topdown parser for our `smengels2` example we have to enter:

```
eag-compile -td smengels2
eagcc smengels2_topdown
```

resulting in an executable named `smengels2_topdown`. Leftcorner parsers are obtained by adding the flag `-lc` to the command line at generation time. This flag is default.

There is a great difference between the two kinds of parsers. Topdown parsers tend to be smaller in size, while leftcorner parsers tend to be a little bit faster then topdown parsers. However topdown parsers can not cope with *leftrecursive* grammars. We call a grammar leftrecursive if there is a nonterminal and a sentential form derived from that nonterminal in one or more steps, starting with the same nonterminal. For such a grammar the EAG compiler will report an error message, when trying to generate a topdown parser.

2.4 Semiterminals

As we can see from the provious example it is quite tedious if we have to describe a syntax rule for a letter in the following way:

```
letter: "a"; "b"; "c"; "d"; "e"; "f"; "g"; "h"; "i";
        "j"; "k"; "l"; "m"; "n"; "o"; "p"; "q"; "r";
        "s"; "t"; "u"; "v"; "w"; "x"; "y"; "z".
```

A shorthand for the right hand side is therefore available, called a semiterminal:

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

This set notation indicates that `letter` is one of the characters in the set. If a set consists of a number of consecutive characters, a more consise shorthand is available:

```
letter: {a-z}.
```

A semiterminal can have several features. Here is an overview:

- `{....}` matches a character of the set.
- `{....}*` matches any sequence of zero or more characters of the set.
- `{....}+` matches any sequence of one or more characters of the set.
- `{....}*!` matches the longest sequence of zero or more characters of the set.
- `{....}+!` matches the longest sequence of one or more characters of the set.
- `^{....}` matches any (single) character not in the set.
- `^{....}*` matches any sequence of zero or more characters not in the set.
- `^{....}+` matches any sequence of one or more characters not in the set.
- `^{....}*!` matches the longest sequence of zero or more characters not in the set.
- `^{....}+!` matches the longest sequence of one or more characters not in the set.

Usually a character in a semiterminal (or a terminal) denotes itself. Some special characters can also be denoted by using an escaping mechanism similar to that used in C:

notation	denotes
`\n`	newline
`\t`	tab
`\-`	minus
`\\`	backslash
`\"`	quote
`\}`	right brace

This same escaping mechanism can also be used in terminals.

The following example is a grammar for expressions (It can be found in the example directory as `expr.eag`). It also indicates more or less a preferred style of writing EAGs:

```
start:
   layout, expr.

expr:
   expr, plus symbol, term;
   expr, minus symbol, term;
   term.

term:
   term, times symbol, factor;
   term, divides by symbol, factor;
   factor.

factor:
   minus symbol, factor;
   plus symbol, factor;
   unsigned factor.

unsigned factor:
   identifier;
   number;
   left parenthesis, expr, right parenthesis.

identifier: {a-z}, {a-z0-9}*!, layout.
```

```
number: {0-9}+!, layout.

plus symbol: "+", layout.
minus symbol: "-", layout.
times symbol: "*", layout.
divides by symbol: "/", layout.
left parenthesis: "(", layout.
right parenthesis: ")", layout.

layout: { \n\t}*!.
```

Observe the clear distinction between the rules of the context free syntax and those of the lexical syntax.

2.5 Grammar transformations

In many cases the generated parsers may have bad runtime behaviour, due to backtracking. However the grammar writer may improve the runtime behaviour by transforming the underlying grammar or by carefully rewriting it.

As mentioned earlier topdown parsers generated by the compiler are smaller than leftcorner parsers. A grammar writer could therefore aim at writing a grammar that is topdown parsable. Such a grammar may not contain leftrecursive rules. Luckily (almost) every context free grammar can be transformed into an equivalent context free grammar that does not contain leftrecursive rules. Consider for instance the following syntax rule with k alternatives that are directly leftrecursive and l alternatives, which are not:

```
A: A, rest alt1;
A: A, rest alt2;
  :
A: A, rest altk;
A: altk+1;
  :
A: altk+m.
```

This syntax rule may be replaced by the following two syntax rules, which are not leftrecursive:

```
A: altk+1, B;
A: altk+2, B;
  :
```

```
A: altk+m, B.

B: rest alt1, B;
B: rest alt2, B;
  :
B: rest altk, B;
B: .
```

For example if we apply the above scheme to the syntax rule `expr` of the previous example we obtain the following rules:

```
expr: term, more terms.

more terms: plus symbol, term, more terms;
more terms: minus symbol, term, more terms;
more terms: .
```

The above scheme works for every grammar that only contains directly left-recursive rules. If there are also indirectly left recursive calls, a more general scheme must be used for this transformation [Aho83].

Another technique worthwhile considering is that of leftfactoring. By leftfactoring is meant the sharing of common prefixes of different alternatives of a syntax rule. Consider for instance the following syntax rule:

```
asbc: ap, asbc, "b";
asbc: ap, asbc, "c".
```

By left factoring this rule we obtain:

```
asbc: ap, asbc, rest asbc.

rest asbc: "b"; "c".
```

This technique can reduce the runtime complexity. In some cases it may even reduce the exponential behaviour to a polynomial one. For leftcorner parsers, nonterminals are automatically leftfactored by the parser generator.

2.6 The commit operator

The commit operator may be used to indicate that whenever a topdown parse succeeds beyond a certain point in an alternative, indicated by this commit operator (`->`), the alternatives succeeding this alternative may be ignored by the parser. This operator is extremely useful to speed up a topdown parser, whenever the grammar writer is sure that two alternatives of a rule are mutually exclusive.

Thus, the commit operator in:

```
apbc: ap1, ->, bc;
apbc: ap2, bc.
```

guarantees that whenever the parser succeeds beyond the member `ap1`, the second alternative will not be tried.

The advantages of the commit operator are that it may be used to speed up the generated topdown parsers or to make a generated parser more deterministic. The disadvantage is that the order of the alternatives of a syntax rule is now significant.

The commit operator is only implemented for topdown parsers. It is ignored if a leftcorner parser is generated.

Chapter 3

Affix grammars

3.1 Introduction and syntax

Thus far we could only generate simple recognizers, which could only indicate whether a piece of text belonged to the given (context free) grammar or not. In this chapter we will augment syntax rules in EAG by giving them parameters, called *affixes*. In an EAG affixes represent *values* that are dynamically evaluated and propagated during the parsing process. These values may be used for instance to produce output for successfull parses. Affix propagation may also steer the parsing process. In this way context sensitive restrictions may be specified in an EAG.

These parameters are added to syntax rule definitions and calls by so called *displays*. A display contains one or more *affix positions*, separated by commas and surrounded by parentheses. Displays may occur anywhere in the left hand side of a syntax rule or in calls. For example:

```
(>i) concatenated with (>j) is (i+j>)
```

The EAG compiler will collect all positions belonging to a nonterminal. Only the order in which affix positions occur is significant. Thus the two following lines have the same semantics:

```
boolean (bool) denotation ("1")
booleandenotation (bool,"1")
```

We define the *arity* of a syntax rule as the number of affix positions it has. A display may also occur after a semiterminal. There are three context conditions:

- All syntax rule elements of a syntax rule must have the same arity.

- A call of a syntax rule must have the same number of positions as the arity of its definition. This also applies to the start rule.

- A semiterminal can only have an arity of zero or one.

An affix position specifies an *affix expression* and an optional *affix flow*. A position is *inherited* if the flow symbol (>) precedes the affix expression. A position is *derived* if the flow symbol follows the affix expression. In previous versions of EAG compilers, the specification of affix flow was obligatory and indicated the dataflow of the affix values. In the current EAG compiler this specification may be omitted, since affix values are propagated at runtime by a data driven propagation mechanism. There is one exception to this rule namely in the specification of predicates, which we will discuss later. Although flow specification may be omitted while writing an EAG, it may make the grammar more readable.

An affix expression consists of one or more *affix terms*, separated either by the concatenation symbol (+), the composition symbol (*) or the union symbol (|). An affix term may be an *affix variable*, an affix constant (a terminal, a number or an element of a finite lattice) or a set. Some examples of affix expressions are:

```
"-" + flength
"dyop" * op * a * b
a single affix variable
20 + length
male | female
```

3.2 Simple transducers

During a parse of a generated parser, affixes may obtain values (and lose them while backtracking). Affix values are of four possible types:

- string

- int

- composed (tuple)

- finite lattice

Affix values originate from affix terminals (constant values), semiterminals, predicates and meta rules. Affix values are propagated by a data driven propagation and restriction mechanism. Predicates, meta rules and finite lattices will be discussed later.

In our first example affix values are only derived from affix constants and semiterminals. When a string is recognized by a semiterminal with a display, the string recognized is assigned to the affix expression in this display. If this expression is a single affix variable, this amounts to assigning the string just read to this variable and unassigning it as the parser backtracks over the semiterminal.

Affix values of the string domain may be concatenated into a new affix value by the concatenation operator `+`. When a parse is found, the values of the non inherited affixes of the start rule are printed (on stdout). Our first example is a simple transducer that illustrates the concatenation of string values (`dupl.eag`):

```
duplicate (id + " + " + id + " = " + id + id +"\n">):
   identifier (id>).

identifier (l + ls>):
   {a-z} (l>), {a-z0-9}*! (ls>), { \t\n}*!.
```

If we compile the above grammar and execute the generated parser with input `krom` the following output is produced:

```
krom + krom = kromkrom
```

When the concatenation operator is applied to integer affix values, they are added together. Consider for instance the following example which outputs the number of identifiers in the input: (`countids.eag`):

```
count (out).

count (1 + ct>):
   identifier (id), count (ct>);
count (0>):.

identifier (l + ls>):
   {a-z} (l>), {a-z0-9}*! (ls>), { \t\n}*!.
```

The *head grammar* of an EAG is formed by removing all displays. We will call an EAG a *transducer* if it recognizes the same language as its head grammar.

3.3 Affix propagation

While parsing, a syntax tree is built, which is decorated with affix values. For every syntax tree node, the corresponding affix positions are built plus links between affix nodes along which their values will be propagated. At runtime, affix flow is data driven, that is: whenever an affix gets a value, affix propagation will try and propagate this value to positions that depend on this affix. This process continues until nothing more can be propagated. Parsing then resumes.

Affix positions occurring in the left hand side of a syntax rule are built before a parse is tried according to that syntax rule. Affix positions occurring in the right hand side of a syntax rule are built when the surrounding member is parsed. Affix propagation is therefore likely to occur at this point.

One may see an affix position as a gateway between values propagating from the left hand side of a syntax rule and the values that must be propagated from the call of that syntax rule occurring in some right hand side. For this reason, affix positions belonging to a syntax tree node are split in an upper and a lower side. In the upper side a representation is stored of the affix expression occuring in the call of this syntax rule. In the lower side a representation of the corresponding left hand side position is stored. The combination of the upper and lower side of a position is also called an *affix position slice*. For both sides a numeric variable, called the *sill*, is maintained that gives the number of affixes in that side of the slice that do not have a value yet. Whenever the sill of a side drops to zero, the corresponding affix expression is calculated and propagated to the other side of the slice.

Affix propagation may be traced just like the parsing process. For instance if we compile our first example as follows:

```
eag-compile -T -td dupl
eagcc dupl_topdown
```

and give it input `ko`, we obtain a trace of the parse and affix propagation. A part of it is listed here:

```
2 '\e': tracing position 0 of rule_duplicate in module dupl
2 '\e': lo = (sill = 0, a_id<"ko"> + a_gen_0<" + "> + a_id<"ko">
                     + a_gen_1<" = "> + a_id<"ko"> + a_id<"ko">
                     + a_gen_2<"\n">)
2 '\e': hi = (sill = 1, a_startrule_0)
2 '\e': tracing position 0 of rootnode in module dupl
2 '\e': lo = (sill = 0, a_startrule_0<"ko + ko = koko\n">)
2 '\e': hi = (sill = 0, _)
```

```
A parse was found
ko + ko = koko
1 'k': <rule_identifier
0 'k': <rule_duplicate
```

Affix propagation is traced per position that is being propagated. It always consists of three lines. The first indicates the position that is being propagated. The second indicates the sill and affix expression on the lower side of the slice. As we can see, if an affix has a value, this value is printed after the affix between brackets. The third line gives the same for the upper side of the slice.

3.4 Splitting of affix values

When an affix value is propagated to a position that is specified as the concatenation of several affixes, this value is split nondeterministically over these affixes. Every possible split will be tried (which can easily lead to many parses). Consider for instance the following grammar (`minus2.eag`):

```
start (div>).

start (div>):
   layout, as (div+"aa">).

as (tt>):
   {a}+! (tt>), layout.

layout:
   { \n\t}*!.
```

As we can see, this grammar will recognize a string of `a`'s in the input and propagate this string via the affix `tt`, which is then split into two. Since the second affix is a terminal only that split succeeds that propagates `aa` to that terminal. In effect, this grammar outputs two `a`'s less than its input string. If this affix terminal had been an affix nonterminal, any possible split would be tried, resulting in $n + 1$ possible parses if the input string had contained n `a`'s.

3.5 Consistent substitution

The notion of consistent substitution is derived from the same notion in two level Van Wijngaarden grammars. By consistent substitution is meant that each of the occurences of an affix nonterminal in an alternative represents the same affix value. Consistent substitution allows the specification of context sensitive restrictions on the grammar.

Consider for instance the following grammar (`evenas.eag`):

```
start (div>).

start (div>):
   layout, header (id>), as (div+div>), trailer (id>).

as (tt>):
   {a}+! (tt>), layout.

header (id>):
   "BEGIN", layout, identifier (id>).

trailer (id>):
   "END", layout, identifier (id>).

identifier (id>):
   {a-z}+! (id>), layout.

layout:
   { \n\t}*!.
```

This grammar imposes two context conditions. The first is that the identifier following the `BEGIN` is the same as the one following the `END`. The second condition is caused by the split of the affix value into two equal parts.

Another nice example is the following analyzer of unary fractions (`frac.eag`). It illustrates arbitrary flow and consistent substitution in EAG:

```
numeral (value+nlcr):
   layout, rational (value), layout.

rational (whole value + "+" + fractional value):
   number (whole value, whole length, ""), ".",
      number (fractional value, fractional length,
              "-" + fractional length).
rational (value):
   number (value, length, "").

number (digit value + "+" + number value, "1" + length, suffix):
```

```
   digit (digit value, length + suffix),
      number (number value, length, suffix).
number (digit value, "1", suffix):
   digit (digit value, suffix).

digit ("11^(" + exponent + ")",  exponent): "1".
digit ("0",                      exponent): "0".

layout: { \t\n}*!.
```

3.6 Composition

The composition operator `*` allows the grammar writer to build composite affix values which may be used for instance to build complex datastructures while parsing. Composition is not associative nor commutative. A possible usage of composed affix values is building an explicit parse tree while parsing. It may then be analyzed later by predicates (see next section). This enables the EAG writer to specify multi pass translators in EAG.

By their definition it is only possible to specify 2-tuples, 3-tuples and so on. To facilitate building linked lists or trees in EAG, a (builtin) affix is provided, called `nil`, whose value is a tuple with zero fields.

If a non inherited affix of the start rule is of type tuple, a text representation of the corresponding affix value is printed on a succesful parse.

An example is the following EAG that recognizes an expression, while building a syntax tree for it (`expr2.eag`).

```
start (e>):
   layout, expr (e>).

expr ("dyop" * "+" * e * t>):
   expr (e>), plus symbol, term (t>);
expr ("dyop" * "-" * e * t>):
   expr (e>), minus symbol, term (t>);
expr (t>):
   term (t>).

term ("dyop" * "*" * t * f>):
   term (t>), times symbol, factor (f>);
term ("dyop" * "/" * t * f>):
   term (t>), divides by symbol, factor (f>);
term (f>):
```

```
      factor (f>).

   factor ("monop" * "-" * f>):
      minus symbol, factor (f>);
   factor ("monop" * "+" * f>):
      plus symbol, factor (f>);
   factor (f>):
      unsigned factor (f>).

   unsigned factor ("id" * id>):
      identifier (id>);
   unsigned factor ("num" * n>):
      number (n>);
   unsigned factor (e>):
      left parenthesis, expr (e>), right parenthesis.

   identifier (l +ls>):
      {a-z} (l>), {a-z0-9}*! (ls>), layout.

   number (n>):
      {0-9}+! (n>), layout.

   plus symbol: "+", layout.
   minus symbol: "-", layout.
   times symbol: "*", layout.
   divides by symbol: "/", layout.
   left parenthesis: "(", layout.
   right parenthesis: ")", layout.

   layout: { \n\t}*!.
```

3.7 Predicates

A *predicate* is a syntax rule that will only produce empty if a certain context condition holds. If this is not the case, it will not produce anything. Predicates are builtin or defined in terms of other predicates. The evaluation of a predicate may either succeed, that is recognize nothing, or it may fail depending on the values of its affixes. By evaluating predicates while parsing, *affix directed parsing* is obtained.

Two particularly useful builtin predicates are `equal`, which only succeeds if its two affix positions equal and and `not equal`, which only succeeds on inequality.

Predicates are typically used to impose context conditions on a certain context free grammar. One may think for instance of the grammar for a programming language. Predicates may be used to enforce that for every application of an identifier an appropriate declaration is present or that every identifier is only declared once. An example of such an EAG is the following grammar. It also illustrates the specification of linked lists in EAG (`decl.eag`):

```
simple:
   simple header,
      declarations (decls>),
      applications (>decls),
      simple trailer.

simple header:
   "BEGIN", layout.

simple trailer:
   "END", layout.

declarations (decls>):
   "DECLARE", layout, identifier list (decls>).

identifier list (decls>):
   identifier (id>), more identifiers (>id * nil, decls>).

more identifiers (>old decls, new decls>):
   ",", layout,
      identifier (id>), not in list (>id, >old decls),
      more identifiers (>id * old decls, new decls>);
more identifiers (>decls, decls>):.

applications (>decls):
   "APPLY", layout, applied id list (>decls).

applied id list (>decls):
   identifier (id>), in list (>id, >decls),
      more applied ids (>decls).
```

```
more applied ids (>decls):
   ",", layout,
      identifier (id>), in list (>id, >decls),
      more applied ids (>decls);
more applied ids (>decls):.

not in list (>ls, >nil):.
not in list (>ls, >string * list):
   not equal (>ls, >string), not in list (>ls, >list).

in list (>ls, >ls * tuple):.
in list (>ls, >string * tuple):
   not equal (>ls, >string), in list (>ls, >tuple).

identifier (ls>):
   {a-z}+! (ls>), layout.

layout: { \n\t}*!.
```

3.8 Delayed affix evaluation

Since predicates do not recognize any input they depend for their execution solely on the values of their affixes. It is clear that right hand sides of predicates may only be evaluated if the values of certain affixes are known. Consider for instance the predicate `in list` in the previous example. The right hand sides of this predicate can only be evaluated if both of the affix positions have got a value. We will call affix positions whose values must be known for the evaluation of a predicate *critical affix positions*. We mark an affix position as critical by marking this position in the left hand side of a predicate as an inherited one. As mentioned earlier this is the only location in an EAG, where specification of flow is necessary.

A predicate's right hand side is only evaluated if the sills of the upper sides of all of its critical affix positions has dropped to zero. Since this can occur much later in the parsing process, the upper sides of critical affix positions are marked as *delayed*, when this predicate is executed for the first time. If, during affix propagation, the sills of the upper sides of the critical positions become zero, these positions are unmarked and evaluation of the right hand sides of the predicate commences. Right hand sides of predicates are always evaluated topdown. The termination of this evaluation must be ensured via the affix values of critical affix positions.

If a predicate does not have any critical affix position it is called a *semipredicate*. Semipredicates need not be delayed and are always immediately evaluated. Semipredicates are either builtin for special purposes or used by the grammar writer to initialize some datastructures. Three builtin semipredicates are very often used namely `end of sentence`, `column (int>)` and `row (int>)`.

The semipredicate `end of sentence` succeeds only if the input pointer has reached the end of file. It is typically used in an EAG in which the input is parsed while building a datastructure, which is transformed later into output when all of the input has been recognized using predicates. It is clear that this latter transformation is only wanted when all of the input has been parsed. However it may sometimes occur that a prefix of the input conforms to the grammar, which would enable to start transformation of the already built datastructures prematurely (and backtracking again when the final check on end of file is met). It is then beneficial for the runtime behaviour of the resulting parsers to specify that end of file must be reached before such a transformation starts. The next example will show an application of this.

The two semipredicates `column` and `row` will return the column and line number respectively of the current position in the input.

The delayed affix evaluation mechanism may for instance be used to specify grammars for programming languages in which an identifier may be used before it is declared. Consider for instance the following grammar for such a programming language, which transforms its input into a program in a C-like language (`lazy.eag`):

```
start (out>).

start ("program\n\t{\n"+cdecls+capplies+"\t}\n">):
   layout, begin symbol,
      units (>decls, >nil, >nil, decls>, applies>),
      end symbol, endofsentence,
      code decls (>decls, cdecls),
      code applies (>applies, capplies).

units (>alldecls, >olddecls, >oldapplies, newdecls>, newapplies>):
   unit (>alldecls, >olddecls, >oldapplies, idecls>, iapplies>),
      semicolon,
      units (>alldecls, >idecls, >iapplies, newdecls>, newapplies>).
units (>alldecls, >olddecls, >oldapplies, newdecls>, newapplies>):
   unit (>alldecls, >olddecls, >oldapplies, newdecls>, newapplies>).
```

```
unit (>alldecls, >olddecls, >applies, newdecls>, applies>):
   declaration (>olddecls, newdecls>).
unit (>alldecls, >decls, >oldapplies, decls>, newapplies>):
   application (>alldecls, >oldapplies, newapplies>).

declaration (>decls, decls * ls>):
   decl symbol, identifier (ls>), not in list (>ls, >decls).

application (>alldecls, >applies, applies * ls>):
   apply symbol, identifier (ls>), in list (>ls, >alldecls).

not in list (>ls, >nil):.
not in list (>ls, >list * string):
   not equal (>ls, >string), not in list (>ls, >list).

in list (>ls, >tuple * ls):.
in list (>ls, >tuple * string):
   not equal (>ls, >string), in list (>ls, >tuple).

code decls (>nil, empty>):.
code decls (>list * ls, cdecls+"\t  decl "+ls+";\n">):
   code decls (>list, cdecls>).

code applies (>nil, empty>):.
code applies (>list * ls, capplies+"\t  apply "+ls+";\n">):
   code applies (>list, capplies>).

begin symbol: "BEGIN", layout.
end symbol: "END", layout.
semicolon: ";", layout.
decl symbol: "DECL", layout.
apply symbol: "APPLY", layout.

identifier (ls>):
   {a-z}+! (ls>), layout.

layout:
   { \n\t}*! (ignore>).
```

Chapter 4

Meta rules

4.1 Introduction and syntax

Meta rules are used to impose restrictions on the values of affixes. These restrictions are specified by means of a context free grammar, called the *meta grammar* of the EAG.

A meta rule consists of a *meta nonterminal* followed by a double colon (::), followed by one or more *meta alternatives*, separated by semicolons, ending in a period. Meta nonterminals are identifiers not ending in an integer suffix. A meta alternative consists of an optional affix expression. Examples:

```
gender :: male | female | neutral.

number :: 1; 2; 3.

expr :: dyop * op * expr * expr;
        monop * op * expr;
        factor.

preamble :: "void main (argc, argv)" + nlcr +
            " int argc;" + nlcr +
            " char **argv;" + nlcr +
            "\t{".

empty ::.
```

There are a number of context conditions, to which the meta grammar must comply:

- Identifiers occurring in union expressions in the right hand side of a meta rule, for which there is no meta rule with the same name, introduce elements of a finite lattice.

- For all other identifiers occurring in the right hand side of a meta rule there must be a corresponding (possibly builtin) meta rule with the same name.

- All meta alternatives of one meta rule should have the same type (string, int, tuple or lattice). An empty alternative has type string and describes the empty string.

4.2 Meta rules and affix propagation

We call an affix nonterminal, for which there exists a meta rule with the same name *minus a possible integer suffix* a *meta defined* affix nonterminal. Thus `mode` and `mode2` may be used as two different instances of the same meta rule `mode`. A meta defined affix nonterminal can only have a value that is in the language described by the corresponding meta rule. An affix nonterminal that is not meta defined is called a *free* affix nonterminal and may therefore have any value. Whenever a value is propagated to an affix that is meta defined a check is made whether this value is admissable.

We can distinguish two kinds of meta rules by the size of the produced language, namely meta rules that correspond to a language with one element and meta rules that correspond to a language with more than one element. If the language of a meta rule contains only one element (such as the meta rule `preamble` in the above examples) this element is computed at compile time by the EAG compiler and hardcoded in the generated parser. The previously mentioned checks at propagation time are then omitted, since the value carried by these affix nonterminals is by default in the language of the corresponding meta rules. These meta rules are typically used to introduce names for affix constants.

For the other two kinds of meta rules, recognizers are generated by the compiler which are called at propagation time. For a meta rule of type tuple, a recognizer is generated that checks the structure of the corresponding composed value at propagation time. For meta rules of the other two types a *topdown* parser is generated (with some heuristics) to check if an affix value is in the language described by the meta rule. This means that meta rules of the string or integer type may not be leftrecursive.

4.3 Finite lattices and affix restriction

Union affix expressions serve to introduce finite lattices. The domain of such affixes is the powerset of the set of all elements belonging to the lattice. Consequently the values that these affixes may take are sets instead of single values. This construction, first introduced in AGFL (Affix Grammars over Finite Lattices), is often used in describing natural languages and reflects the gradual increase in knowledge while parsing. Consider the following example (`simpfin.eag`):

```
PERS :: first | second | third.

NUMBER :: singular | plural.

simple sentence (NUMBER, PERS):
   layout, pers pron (NUMBER, PERS),
      to be (NUMBER, PERS), adjective.

pers pron (singular, first): "I", layout.
pers pron (plural, first): "we", layout.
pers pron (NUMBER, second): "you", layout.
pers pron (singular, third):
   "he", layout;
   "she", layout;
   "it", layout.
pers pron (plural, third): "they", layout.

to be (singular, first): "am", layout.
to be (NUMBER, second): "are", layout.
to be (singular, third): "is", layout.
to be (plural, first | third): "are", layout.

adjective: "great", layout.
adjective: "small", layout.

layout: { \t\n} *!.
```

Affixes take the subset corresponding to their indicated domain as initial value. In this case the affix `NUMBER` is given the set `{singular, plural}` as initial value. Furthermore we follow the AGFL convention, that elements may be used in syntax rules to denote their corresponding singleton sets. Union expressions may also occur in syntax rules: the EAG compiler will evaluate them and replace them by the corresponding subsets in the affix domain.

Consistent substitution demands that all affixes with the same name in an alternative have the same value. This means that whenever a value gets propagated to such an affix, it is intersected with the value of that affix. If this results in the empty set, the parse is rejected. If it results in a smaller set, this smaller set is propagated to all occurrences of the affix in the alternative. Thus the actual propagation mechanism for affixes over finite lattices is set restriction instead of value propagation. Since all affixes over finite lattices get a single initial value and their value may only become smaller during the parsing process, no recognizer routines are necessary for these meta rules.

Values resulting from the parsing process need not be singleton sets. For instance, after parsing the sentence `"you are great"`, the affix `NUMBER` will still have the set `{singular, plural}` as value, indicating that the sentence gives insufficient information to decide if it is used in singular or plural form. When the start rule of the grammar has affixes over lattices, a text representation of the resulting sets is printed on a succesfull parse.

4.4 Meta rules as abstraction

One of the very first lessons in software engineering dictates that programs should never contain unnamed constants. The same holds for EAG. When an affix constant is very large or when an affix constant is used at several places in an EAG, it is useful to introduce a name for it by means of a meta definition. Consider for instance the following example, that codes a (rather silly) C program out of a list of identifiers (`meta1.eag`):

```
program (preamble + prints + postamble):
   layout, identifiers (prints).

preamble ::
        "#include <stdio.h>" + nlcr + nlcr +
        "main ()" + nlcr + tab + "{ ".

postamble ::
        "exit (0);" + nlcr + tab + "}" + nlcr.

identifiers (prints + print header + id + print trailer):
   identifiers (prints), identifier (id).
identifiers (empty):.
```

```
print header :: "printf (" + quote.
print trailer :: "\\n" + quote + ");" + nlcr + tab +"  ".

identifier (l + ls):
   {a-z} (l), {a-z0-9}*! (ls), layout.

layout: { \t\n}*!.
```

In this example we can observe three builtin meta defined affix nonterminals namely `nlcr`, `quote`, and `tab` whose value should be self exemplifying.

4.5 Meta rules and sets

Sets may also occur in the right hand side of a meta rule. In this way one can easily specify a regular expression to which affix values must comply. Consider for instance the following example (`meta2.eag`):

```
start (specid + nlcr):
   layout, id (specid).

specid :: ^{k}*! + "k" + ^{k}*! + "k" + ^{k}*!.

id (l + ls):
   {a-z} (l), {a-z0-9}*! (ls), layout.

layout: { \t\n}*!.
```

In this way we can easily specify the intersection of two context free grammars.

A grammar writer may also specify sets in affix positions. For these sets anonymous meta rules are generated by the compiler, which will check propagated affix values at runtime.

4.6 Meta rules for describing datastructures

Meta rules may also be used to specify datastructures, which may be built while parsing and transformed after parsing. Such a specification of composite affix values serves the readability of the grammar. It may also speed up the splitting of composite affix values, because the affix values resulting from a split can be checked. A small example is the following description (`meta3.eag`):

```
start (xout>):
   layout, begin symbol, units (out>), end symbol,
   endofsentence, xformunits (>out, xout>).

units (unit * units>):
   unit (unit>), semicolon symbol, units (units>);
units (unit>):
   unit (unit>).

unit (ident * id>): identifier (id>);
unit (number * int>): number (int>);
unit (skip>): skip symbol.

xformunits (>unit * units, xunit + xunits>):
   xformunit (>unit, xunit>), xformunits (>units, xunits>);
xformunits (>unit, xunit>):
   xformunit (>unit, xunit>).

xformunit (>ident * string, string + ";"+ nlcr>):;
xformunit (>number * int, string + ";" + nlcr>):
   int to string (>int, string>);
xformunit (>skip, empty>):.

unit :: ident * string;
        number * int;
        skip.

skip :: "skip" * nil.
ident :: "identfier".
number :: "number".

identifier (ls>):
   {a-z}+! (ls>), layout.

number (int>):
   {0-9}+! (digits>), layout, string to int (>digits, int>).

begin symbol: "BEGIN", layout.
end symbol: "END", layout.
skip symbol: "SKIP", layout.
semicolon symbol: ";", layout.
```

```
layout: { \n\t}*!.
```

4.7 Meta rules and typing

Not only meta rules but every affix and affix position must have one of the four basic types (string, int, tuple (composed) or finite lattice) or have a polymorfic type (nonlat or any). Polymorfic affixes may be passed on, but they may not be manipulated. Polymorfic affixes can only be introduced by the builtin metarule `nonlat`, indicating a non lattice type.

The EAG compiler will try and infer the types of the affixes and the positions. Sometimes, it may occur that the compiler can not infer the type of an affix. In those cases it will give an error message. Such is for instance the case in the following piece of a grammar.

```
not in list (>ls, >nil):.
not in list (>ls, >ls2 * list):
   not equal (>ls, >ls2), not in list (>ls, >list).
```

From the affix `nil`, the compiler can infer that the second position of this predicate has type tuple. Consequently it can infer from the call of `not in list` in the second alternative that the affix `list` must also have type tuple, because the corresponding affix positions must have the same type.

From the usage of a predicate or syntax rule the compiler can often infer the type of the positions. Suppose that it can indeed infer that the first affix position has type string. Consequently the affix `ls` in both alternatives must also have type string.

There is, however, no way by which it may infer that the affix `ls2` in the second alternative has type string (`not equal` is a polymorfic predicate). We may correct this example in the following way, using the builtin meta rule `string`:

```
not in list (>ls, >nil):.
not in list (>ls, >string * list):
   not equal (>ls, >string), not in list (>ls, >list).
```

For typing purposes, four builtin meta rules are provided namely `int`, `string`, `tuple` or `nonlat`.

Chapter 5

Syntax directed editors

5.1 Introduction

In the previous chapters we discussed the usage of the EAG formalism in the specification of recognizers and translators. In this chapter we discuss using EAG in language prototyping. Whenever a new language must be developed, the developer will want to try several possibilities for certain syntactical constructs. In such a case it can be very handy to have a tool to check whether the language under development is easy to use. A *syntax directed editor generator* is such a tool.

A syntax directed editor is an editor, that has knowledge of the language for which it was created. Thus it can automatically flag syntax errors and sometimes also semantical errors. There are two different kinds of syntax directed editors, namely *template editors* and *text editors*. Some syntax directed editors combine the features of both. They are called *hybrid editors*.

Template editing is based on using *placeholders* and *templates*. A placeholder is a special symbol representing a certain syntactic construct that is not yet defined. Usually they are derived from the nonterminals of the corresponding grammar. A template is a frame of a syntactic construct, usually derived from a right hand side of the grammar, in which the nonterminals are replaced by their corresponding placeholders. The user actions in such an editor typically consist of focussing on a placeholder and then replacing it by one of the templates derived from the corresponding right hand sides. The advantage of a template editor is that it is impossible to enter syntactically incorrect descriptions. Its main disadvantage is that entering small constructs (simple assignations, expressions, etc.) is quite tedious. It is also difficult to change a text at a number of places at the same time.

A text editor is an editor in which the text is syntactically checked after the user has finished editing. Usually after such a check an unparse is done of the corresponding syntax tree. With such an editor the user has the liberty of changing large amounts of text in one action. Its disadvantage is that the user is only shown the new structure of the syntax tree after parsing.

The EAG compiler may also be used to generate a hybrid syntax directed editor from an EAG. Such a syntax directed editor uses the X11 window system for its user interface. Two libraries are also available in the EAG compiler package to support the runtime system of generated syntax directed editors.

When using such a syntax directed editor, a user has the following choice of actions:

- Selecting a syntactic piece of the text, which will then be highlighted. This piece of text is called the *focus*.
- Enlarging the focus to its father node in the syntax tree.
- Editing the focus. Upon finishing editing, the text in the editors text buffer will be reparsed. The syntax tree resulting from this action is then unparsed in the editors text buffer, thus reflecting the changes in the syntax tree.
- Replacing a focussed placeholder by a syntactically correct template. The text resulting from this action is also parsed and unparsed. (Although we know aforehand that the corresponding text is syntactically correct, we must still unparse because of layout requirements).
- Reading and writing of the text buffer from or to a file.
- Edit the rules by which a recognized text is unparsed.

5.2 Placeholders

As mentioned above, placeholders represent syntactic constructs that are not yet defined. In the syntax directed editors generated by the EAG compiler, two kinds of placeholders exits, namely the *typed* and the *untyped* placeholders. A typed placeholder replaces a specific syntactic construct. It is denoted by the name of the corresponding nonterminal between two special braces (`<|` and `|>`). For example:

```
<|expression|>
```

An untyped placeholder does not represent any specific syntactic construct. It is denoted by `<|>`.

We may view the addition of placeholders as an extension of the original grammar in which every syntax rule is augmented with two extra syntax rule elements. However, this augmented grammar may become (locally) ambiguous because of the untyped placeholders. Thus it may be possible that the generated parser finds more than one parse for a piece of text containing untyped placeholders, even if the original grammar is not ambiguous. Whenever a parse succeeds, the resulting syntax tree is combined (folded) with those of previously succeeding parses. This folded syntax tree is then unparsed after parsing. However, if an untyped placeholder can be unambiguously parsed, the unparser will replace this untyped placeholder by its corresponding typed one.

5.3 Layout as a special syntax rule

A generated syntax directed editor contains both a parser and an unparser. This unparser is called if a parse of the editors' main text buffer succeeds. The unparser will introduce layout between syntactical constructs by means of a heuristic algorithm. Its heuristics are derived by the EAG compiler from the structure of the grammer. Especially layout will flank placeholders and must be properly parsed. For this reason, the syntax rule that defines layout is treated in a special way by the EAG compiler when generating syntax directed editors. Three restrictions are checked by the compiler concerning the syntax rule defining layout:

- It should have `layout` as name.
- It should have arity zero.
- It may not contain calls of syntax rules.

In practice, this layout rule should always have the following form:

```
layout: { \t\n}*!.
```

5.4 Generating a syntax directed editor

In the coming chapter examples will be based on a very simple language called Pico. Its EAG description is as follows (`pico.eag`):

```
picoprogram:
   layout, "BEGIN", layout,
   declarations, series,
   "END", layout.

declarations:
   "DECLARE", layout, identifierlist, ";", layout.

identifierlist: type, identifier;
identifierlist:
   identifierlist, ",", layout,
   type, identifier.

series: statement;
series: series, ";", layout, statement.

statement: assignation;
statement: ifstatement;
statement: whilestatement.

assignation:
   identifier, ":=", layout, expression.

ifstatement:
   "IF", layout, expression,
   "THEN", layout, series, elsepart,
   "FI", layout.

elsepart:
   "ELSE", layout, series;
elsepart:
   "ELIF", layout, expression,
   "THEN", layout, series, elsepart;
elsepart:.

whilestatement:
  "WHILE", layout, expression,
  "DO", layout, series, "OD", layout.

expression: expression, "=", layout, expression;
expression: expression, "/=", layout, expression;
```

```
expression: expression, "+", layout, term;
expression: term.

term: term, "*", layout, factor;
term: factor.

factor: "(", layout, expression, ")", layout;
factor: identifier;
factor: number;
factor: boolean.

type: "BOOL", layout;
type: "INT", layout.

identifier: {a-z}, letgits, layout.

letgits: { }*!, {a-z0-9}+!, letgits;
letgits:.

number: {0-9}+!, layout.

boolean: "TRUE", layout;
boolean: "FALSE", layout.

layout: { \t\n}*!.
```

We can generate a syntax directed editor for it by entering the following two commands:

```
eag-compile -ed pico
eaged pico_leftcorner
```

The generation of a syntax directed editor creates two files namely a file that contains the parser and interface to the libraries (`pico_leftcorner.c` in the above example) and another file containing a specification of the unparsing rules as heuristically derived from the grammar (`pico.rules`).

5.5 Using a generated syntax directed editor

Before the generated syntax directed editor can be used, its resources must be registered. In the subdirectory `libedt` (and after installation also in the `lib` directory) a resource file named `Editor.ad` should be present. It can be registered by entering the following command:

```
xrdb -m Editor.ad
```

Alternatively this resource file can be placed in the system wide application defaults directory or your private one (Do not forget to strip the suffix).

When a generated editor is executed it will try and parse the contents of the file indicated by its argument. If no argument is given it will parse a text consisting of a single untyped placeholder. The resulting syntax tree is then unparsed in the editors main text buffer. Finally the editor will map itself on the screen.

If we execute the editor generated for the above example without argument, the picture presented on the next page will appear on the screen.

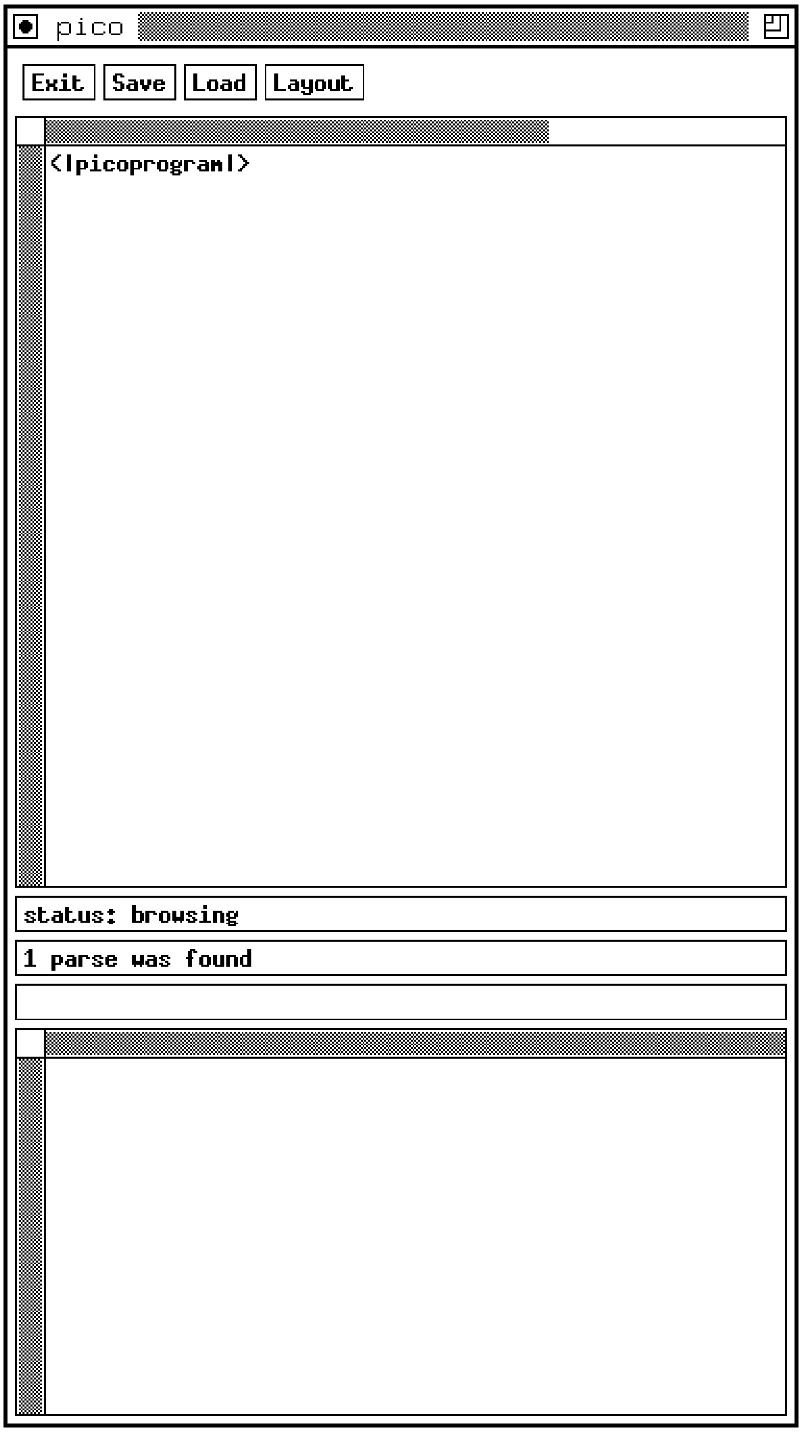

Figure 5.1: Editor in browsing mode

In the top of the application we can observe three command buttons. Below these, the main text buffer is mapped, containing the unparsed syntax tree. The editors text buffer may be scrolled using the scrollbars flanking the text buffer window. Below the text window, the status of the editor is given in three windows. The top one indicates the current mode, which is one of the following three namely *browsing*, *focussed* or *editing*. The second indicates the number of parses it found in the last parse. The third one gives a heuristic error message if the last parse was not succesful. Below these three is the template window. If a user focusses on a typed placeholder, the template window will present every template by which this placeholder may be replaced.

When the editor is in browsing mode, no text is highlighted and no templates are shown. A user may focus on a syntactic construct by moving the mouse to that construct and clicking with the left mouse button. The editor then focusses on the smallest syntactical constuct that contains the location pointed to by the mouse. Thus it is not possible to focus on a part of a terminal or semiterminal nor is it possible to focus on layout. When focussed, the editor will highlight the text of the focus and indicate the mode in the mode window.

The user may enlarge the focus by clicking with the middle mouse button. The focus will then be extended to the father of the current focus.

When focussed on a typed placeholder, the template window will show the templates by which this placeholder may be replaced. A typical situation is shown in the figure on the next page. By clicking with the left mouse button on one of the templates in the template window, the editor will replace the focussed typed placeholder by the selected template. Next it will reparse the entire text and unparse the resulting syntax tree in its main text buffer. Finally it will enter browsing mode.

Whenever the editor is focussed on part of the syntax tree, the user may edit the focus by entering characters: the editor will immediately switch from focussed to editing mode. While editing, the user may insert and delete characters (and newlines) at will as long as they pertain to the focus. As a result of the editing actions the focus will adjust itself to fit. The user may finish editing by entering an escape character (`<ESC>`). The editor will then reparse the text, unparse it again and enter browsing mode.

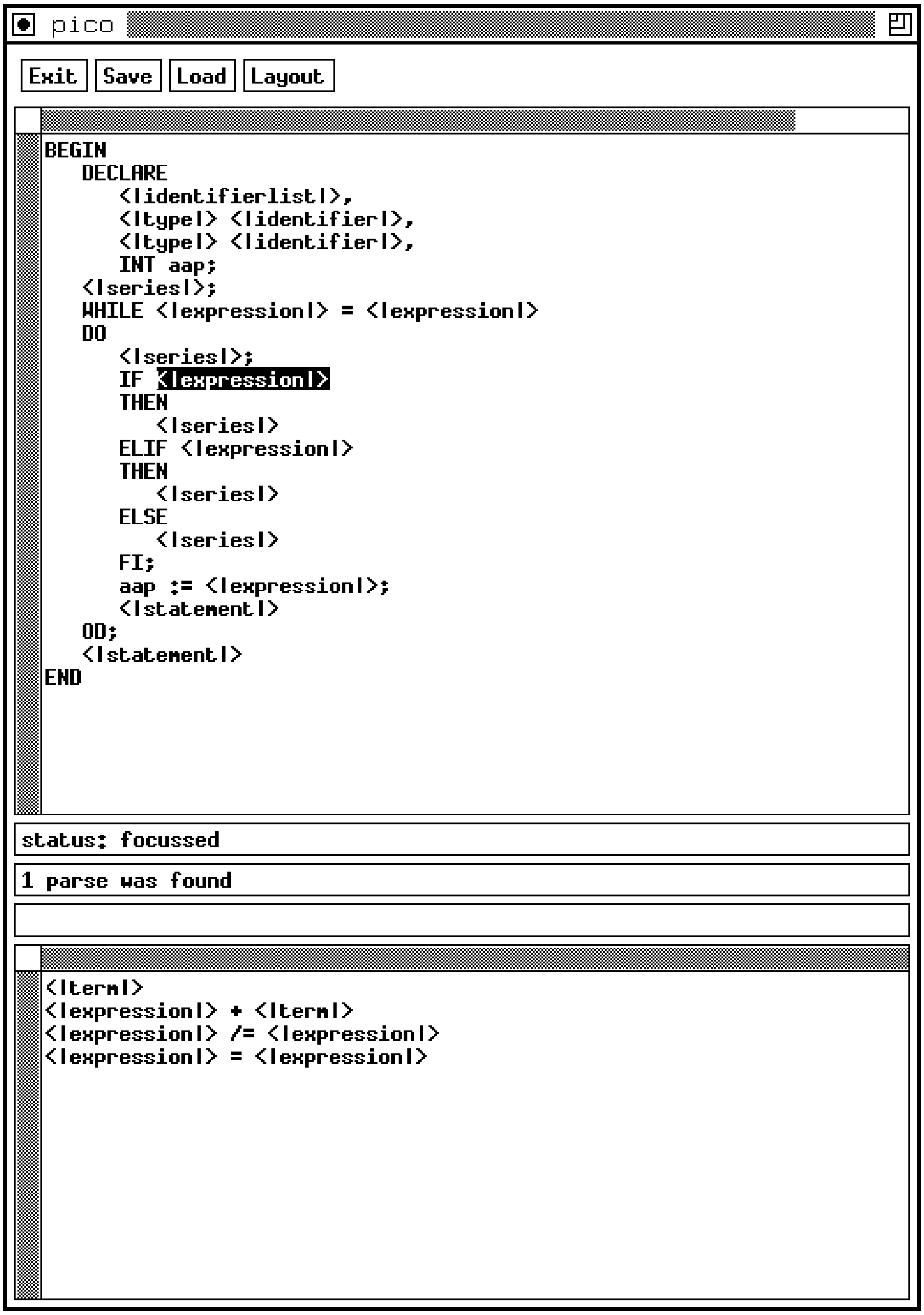

Figure 5.2: Editor in focussed mode

5.6 Editing of unparsing rules

After a reparse of the editors main text buffer, the editor will attempt to unparse the found (possibly folded) syntax tree according to a heuristic algorithm using unparsing rules derived from the original grammar. This algorithm will attempt to layout syntactic constructs in a horizontal way as much as possible. Only if such a horizontal parse can not fit on a line, it will try a vertical parse. However, since the unparsing rules are derived automatically, such an unparse may not always give a pleasantly looking unparse. Especially vertical layout often needs improvement. A user may therefore edit the unparsing rules. After editing an unparsing rule the editors text buffer is again parsed and unparsed, thus reflecting the new unparsing rule. In this way a user may develop a layout preference for the language under development.

To edit an unparsing rule the user must focus upon a construct whose unparsing must be changed. She must then click the command button, labeled with `Layout`. For example, consider focussing on the comma following the `identifierlist` placeholder. If we then click the `Layout` button, the following window will pop up:

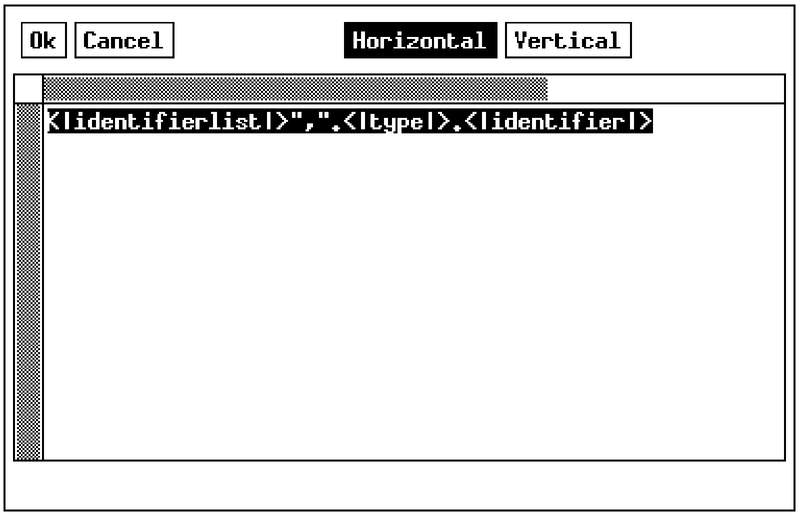

Figure 5.3: Layout popup

With the rightmost buttons one can select between the horizontal and vertical layout rule for the selected construct. The user may only edit the rule by inserting and deleting layout characters and the "`#`" character. This last character may only be placed in front of a placeholder, indicating that the corresponding syntactic construct should always be parsed using vertical layout. There are two space symbols namely the normal space used for shifting placeholders and terminals into the desired column and the ". "symbol, representing a space symbol in the unparsing of the construct. Terminals, placeholders and "." symbols placed in the same column are related to each other. The unparser will unparse the corresponding constructs relative to this column. Clearly, related syntactic constructs only occur in vertical unparsing rules. Consider for example the vertical unparsing of the above construct:

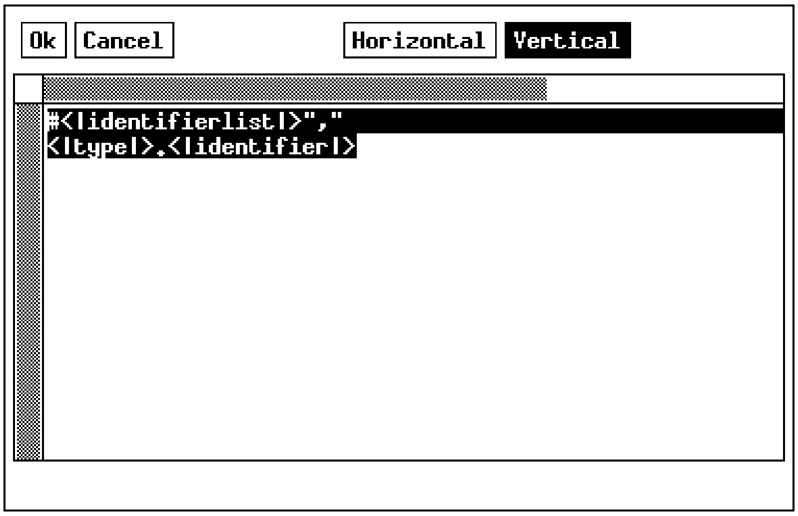

Figure 5.4: Vertical Layout rule

Editing the layout rule ends either by entering the escape (`<ESC>`) character while in the edit window or by clicking the button labeled with `Ok`. By clicking the `Cancel` button, the editing of the layout rule is aborted.

When the syntax directed editor is quitted, the editor will ask to save the unparsing rules if they are changed during the edit session.

Chapter 6

Future Work

The EAG compiler and its runtime system will head off into two seperate directions. Version 2.2 will include a lexicon interface that will considerably improve the usage of EAG for parsing natural languages. Version 3.0 (already partly present in the subdirectory `gen3`) will be directed towards a syntactical (but not semantical) unification of the three affix formalisms EAG, AGFL and CDL3. In particular tuple types will be replaced by the term types already present in CDL3.

However, both versions of EAG will use the same runtime system. They will therefore be provided as two compilers in the same package.

Appendix A

Details of the compiler package

A.1 The package

The following files and directories should be present in the compiler package:

- `Copyright` Copyright notice.
- `Warranty` Warranty notice.
- `Install` Installation notes.
- `History` History of changes to the compiler package.
- `Planned` Planned changes to the compiler package.
- `Makefile` Makefile to make all libraries and the compiler.
- `libalib` An example user extension of the standard library.
- `bin` Directory to install binaries (initially empty).
- `buglist` List of known bugs.
- `doc` Directory containing the source files of this document.
- `examples` Directory containing the examples in this document.
- `gen` Directory containing the source files of the compiler and skeleton generator.
- `include` Directory to install header files (initially empty).
- `lib` Directory to install libraries (initially empty).

`libXedt` Directory containing the source files of the X11 window system support library. In particular it provides two widgets that are used by syntax directed editors.

`libebs` Directory containing the source files of the EAG basic support library.

`libeag` Directory containing the source files of the EAG runtime system.

`libedt` Directory containing the source files of the runtime system necessary for syntax directed editors.

A.2 Installation

To install the EAG compiler and its libraries, edit the `Makefile` in the top directory and adapt the variables `IDIR`, `LDIR`, `BDIR`, `DCFL` and `X11TREE` to suit your local system (The makefile contains two possible suggestions, depending whether you wish to install the compiler locally or system wide). Then enter `make` to create and install the necessary libraries and the compiler.

The software has been tested on the following platforms:

- Sun SPARCstation-20 under Solaris 2.7
- Sun 4/670 under SunOs 4.1.4
- Sun 3/80 under SunOs 4.1.1
- HP 9000/735 under HP-UX 9.0.5
- i686 PC under Linux 2.2.12

A.3 Compiler options

The syntax to use the compiler is:

```
eag-compile [flags] [name]
```

The default extension of EAG is `eag`. Preferably, the name of the file should be given without this extension. The compiler will generate a C file with name `<name>_<type>.c` where `<name>` is the name given as argument and `<type>` is either `topdown` or `leftcorner`, depending on the type of parser wanted. When no input file is given, the compiler will read a grammar from standard input giving it `stdin` as name.

The meaning of the flags is as follows:

- `-h` Provide a short help.
- `-V` Show the current version of the compiler package.
- `-v` Verbose mode. Provide an overview of the passes of the compiler.
- `-fv` Full Verbose mode. Provide an overview of the passes and dump intermediate analysis results on standard error. This option is provided mainly for debugging purposes, since it will create large amounts of output.
- `-p name` Include `name.eag` as extra prelude. This option is mainly used to include user extensions of the standard library.
- `-v1` Accept EAG version 1.6 syntax and semantics. This is a compatibility flag.
- `-ed` Generate a syntax directed editor.
- `-td` Generate a topdown parser.
- `-lc` Generate a leftcorner parser.
- `-lr` Generate a lr parser (not yet implemented).
- `-nl` Do not generate lookahead code. Setting this flag will degrade the performance of the generated parsers. However, the error messages generated when a parse can not be found will improve much.
- `-gl` Generate code to report error messages upon lookahead failure. This will improve the error messages generated when a parse can not be found.
- `-ic` Generate code to interface with C.
- `-il` Do not code any actions on the syntax tree for the syntax rule `layout`. This rule should comply to the rules given in section 5.3. Setting this flag will improve the execution time of generated parsers by some 20%. This flag is implied by the `-ed` flag.
- `-pp` Generate code to parse placeholders. This flag is implied by the `-ed` flag.

-it Generate code to show *indirect templates* too. An indirect template is a template derived from a template that consists of a single typed placeholder by replacing this placeholder by one of its corresponding templates.

-D Generate code to dump the parse tree when a succesful parse is found.

-M Generate code that dumps the number of *matches* found while parsing. A match is the succesful recognition of a terminal. This number is often used by linguists to compare various implementations of EAG or AGFL.

-T Generate code to trace the execution of a parser.

-qc Generate code to check the proper behaviour of the continuation stack. This flag is solely used for debugging purposes.

A.4 Scripts

Four shell scripts are generated by the installation procedure. Their usage is as follows:

```
<script> <name>_<type> [options]
```

where `<script>` is one of these four scripts, `<name>` and `<type>` are as above. The options are only used for user extensions. These scripts are used to compile and link generated parsers or editors with the proper libraries. They will result in an executable with the name `<name>_<type>`. The four scripts and their functions are:

`eagcc` Compile and link a generated transducer.

`eagccl` Only link a generated transducer.

`eaged` Compile and link a generated syntax directed editor.

`eagedl` Only link a generated syntax directed editor.

A.5 The standard library

Before a grammar is parsed by the EAG compiler a prelude containing builtin definitions is parsed (`stddefs.eag`). The contents of this file are as follows:

```
# standard predefines
# meta rules
$ int :: int, recognizer, may produce empty.
$ string :: string, recognizer, may produce empty.
$ tuple :: tuple, recognizer, never produces empty.
$ nonlat :: nonlat, recognizer, may produce empty.
$ nil :: tuple, single, never produces empty.

# standard meta rules
empty ::.
nlcr :: "\n".
quote :: "\"".
back slash :: "\\".
tab :: "\t".

# hyper rules
empty: .
nlcr: "\n".
quote: "\"".
back slash: "\\".
tab: "\t".

# semi predicates: they need not be delayed
$ end of sentence: semipredicate.
$ column (int>): semipredicate.
$ row (int>): semipredicate.

# predicates
$ equal (>nonlat, >nonlat): predicate.
$ not equal (>nonlat, >nonlat): predicate.

$ minus (>int, >int, int>): predicate.
$ mul (>int, >int, int>): predicate.
$ div (>int, >int, int>): predicate.
$ power (>int, >int, int>): predicate.

$ int to string (>int, string>): predicate.
$ string to int (>string, int>): predicate.
$ uppercase (>string, string>): predicate.

$ not in reserved word list (>string, >tuple): predicate.
```

```
$ dump affix (>nonlat): predicate.
```

Definitions that do not start with a dollar sign are merely included and thus hardcoded in generated parsers. For definitions that do start with a dollar sign actual code is present in the EAG runtime library.

For a meta rule in the last category the type, kind and emptiness is specified (The type is one of the basic types. The kind is either single or recognizer. A meta rule may produce empty or not).

A rule that is coded in the runtime library is either a predicate or a semipredicate. Its formal left hand side is specified and typed and its kind is specified.

A.6 User extensions

It is possible for users to extend the standard library with their own libraries. The first step in making a private library is writing its specification which should comply to the same syntax as given for the standard library. Such a specification should have eag as its extension. An example of such a specification can be found in the example extension library (libalib) in the compiler package by the name alib.eag:

```
$ make empty field (>int, >int): predicate.
$ take (>int, >int): predicate.
$ is empty (>int, >int): predicate.
$ show field: semipredicate.
```

The second step to take is to apply the skeleton generator (eag-skel), which is also installed by the installation procedure, to this specification:

```
eag-skel <name>
```

This will result in two files namely <name>.h and <name>.c. The latter gives a template in which the user only has to fill in the C code that implements the actual specification. The code to needed to delay the execution of predicates is already coded by the skeleton generator. Before taking this step it may be very useful to read the file ds.h in the directory libeag of the package which describes the datastructures used by the EAG runtime system. It is also advisable to inspect the file stddefs.c in the same directory to observe how the standard library is implemented.

After filling in the actual code one must compile this C file and construct a small library called lib<name>.a for it. This library should be installed in the lib directory of the package and the files <name>.eag and <name>.h should be placed in the `include` directory. Compilation of an EAG using this user defined library needs adding the flag `-p <name>` for the compilation and the flag `-l<name>` for the C compilation and linking. For example, the file `cactus.eag` in the example user extension library directory `libalib` may be compiled and linked by entering the following two commands:

```
eag-compile -p alib -td cactus
eagcc cactus_topdown -lalib
```

This example also shows the usage of EAG in nondeterministic programming.

Appendix B

A description of the EAG formalism in itself

```
eag:
   layout, rules.

rules:
   rules, rule;
   .

rule:
   start rule;
   syntax rule;
   affix production rule.

start rule:
   call, point symbol.

syntax rule:
   syntax rule group (id), point symbol.

syntax rule group (id):
   syntax rule group (id), semicolon symbol,
      syntax rule element (id);
   syntax rule group (id), point symbol,
      syntax rule element (id);
   syntax rule element (id).
```

```
syntax rule element (id):
   nonterminal (id), colon symbol, alternatives.

alternatives:
   alternatives, semicolon symbol, alternative;
   alternative.

alternative:
   members;
   .

members:
   members, comma symbol, member;
   member.

member:
   call;
   quoted string;
   semiterminal;
   commit symbol.

call:
   nonterminal (id).

semiterminal:
   set, display;
   set.

nonterminal (id):
   nonterminal elements (id), not equal (id, empty).

nonterminal elements (id1 + id2):
   nonterminal elements (id1), identifier (id2);
nonterminal elements (id):
   nonterminal elements (id), display;
nonterminal elements (empty):.

display:
   open symbol, directed affix expressions, close symbol.
```

```
directed affix expressions:
   directed affix expressions, comma symbol,
      directed affix expression;
   directed affix expression.

directed affix expression:
   flow symbol, affix expression;
   affix expression, flow symbol;
   affix expression.

affix production rule:
   affix nonterminal, double colon symbol,
      affix alternatives, point symbol.

affix alternatives:
   affix alternatives, semicolon symbol, affix alternative;
   affix alternative.

affix alternative:
   affix expression;
   .

affix expression:
   affix term, rest affix expression.

rest affix expression:
   plus symbol, affix term, rest concatenation;
   star symbol, affix term, rest composition;
   union symbol, affix term, rest union;
   .

rest concatenation:
   plus symbol, affix term, rest concatenation;
   .

rest composition:
   star symbol, affix term, rest composition;
   .
```

```
rest union:
   union symbol, affix term, rest union;
   .

affix term:
   affix nonterminal;
   quoted string;
   number;
   set.

affix nonterminal:
   affix nonterminal, identifier (name>);
   identifier (name).

set:
   up symbol option, bracket string, set options.

set options:
   plus symbol, strict symbol;
set options:
   star symbol, strict symbol;
set options:
   plus symbol;
set options:
   star symbol;
set options:
   .

up symbol option:
   up symbol;
   .

commit symbol:       "->", layout.
strict symbol:       "!", layout.
open symbol:         "(", layout.
close symbol:        ")", layout.
star symbol:         "*", layout.
plus symbol:         "+", layout.
comma symbol:        ",", layout.
point symbol:        ".", layout.
double colon symbol: "::", layout.
```

```
colon symbol:        ":", layout.
semicolon symbol:    ";", layout.
flow symbol:         ">", layout.
union symbol:        "|", layout.
up symbol:           "^", layout.

identifier (l + ls):
   {a-zA-Z} (l), {a-zA-Z0-9}*! (ls), layout.

number:
   {0-9}+!, layout.

quoted string:
   quote, qchars, quote, layout.

bracket string:
   "{", schars, "\}", layout.

qchars: ^{\"\\}+!, qchars;
qchars: specsym, qchars;
qchars:.

schars: ^{\}\\}+!, schars;
schars: specsym, schars;
schars:.

specsym (>trailer): "\\-";
specsym (>trailer): "\\n";
specsym (>trailer): "\\t";
specsym (>trailer): "\\\\";
specsym (>trailer): "\\\}";
specsym (>trailer): "\\\"";

layout: { \t\n}*!.
```

Bibliography

[Aho83] A.V.Aho, R.Sethi, J.D.Ullman,
Compilers: Principles, Techniques and Tools,
Addison-Wesley Publication, Reading, Massachusetts, 1986

[Bra92] M.G.J. van den Brand,
"PREGMATIC, A generator for incremental programming environments",
Ph.D. thesis, University of Nijmegen, 1992

[Knu68] D.E. Knuth,
"Semantics of context-free languages", in:
Math Systems Theory,
2:2, pp 127-145, 1968

[Kos71] C.H.A. Koster,
"Affix Grammars", in:
J.E.L Peck, ed.
Algol 68 Implementation,
pp 95-109, North Holland Publishing Company, Amsterdam, 1971

[Mei86] H. Meijer,
"Programmar: A translator generator",
Ph.D. thesis, University of Nijmegen, 1986

[Mor89] M.P.G. Moritz,
"Description and analysis of static semantics by fixed point equations",
Ph.D. thesis, University of Nijmegen, 1989

[Seu93] M. Seutter,
"Informal introduction to the Extended Affix Grammar formalism and its compiler",
University of Nijmegen, Sept. 1993

[Wat74] D.A. Watt,
"Analysis-oriented two-level grammars",
Ph.D. thesis, University of Glasgow, January 1974

[Wij76] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster,
M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, R.G. Fisker,
Revised Report on the Algorithmic Language ALGOL 68,
Springer-Verlag, Berlin, 1976.

Contents