GLAMMAR USER MANUAL

VERSION 1.0, JUNE 1991

Eric Voss

Computer and Communication Systems,
Department of Computer Science,
University of Nijmegen,
P.O. Box 9010,
NL-6500GL Nijmegen,
The Netherlands,
Email: ccs@cs.kun.nl

Abstract. This document tells you how to use the GLAMMAR compiler compiler system. This manual assumes you are familiar with context free grammars. It is written for people who intend to use, test or to investigate the usefulness of the GLAMMAR compiler compiler system in a practical environment.

Contents

List of Figures

Copyright

No Warranty

Because GLAMMAR is licensed free of charge, we provide absolutely no warranty, to the extent permitted by applicable law. The University of Nijmegen or other parties provide GLAMMAR "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should GLAMMAR prove defective, you assume the cost of all necessary servicing, repair or correction.

In no event unless required by applicable law will the University of Nijmegen or any other party who may modify and redistribute GLAMMAR as permitted above, be liable to you for damages, including any lost profits, lost monies, or other special, incidental or consequential damages arising out of the use or inability to use (including but not limited to loss of data or data being rendered inaccurate or losses sustained by third parties) the program, even if you have been advised of the possibility of such damages, or for any claim by any other party.

CHAPTER I

Introduction

This manual is intented to be both a user's guide and a reference manual for GLAMMAR — a translator generator language. Chapter II deals with the plain language, starting with simple context free recognizers, ending with more sophisticated Extended Affix Grammars. Chapter III explains some details that are of mostly practical interest. For example, it lists all built in rules, contains the manual page and gives some hints on the subject of debugging. Furthermore it explains how to make efficient deterministic programs. Examples are extensively used throughout this manual as an explanation mechanism. Careful attention to these examples is therefore the key to understanding this manual.

1. Glammar: a compiler compiler

A number of tools have been developed specifically to help construct translators. These tools range from scanner and parser generators to complex systems, variously called *compiler-generators, translator-writing systems* or *compiler compilers*. They all require a specification of the source language and target machine to produce a compiler for the source language that produces code for the target machine. In GLAMMAR the description of both, the source language and the associated target machine code, are specified in the same descriptive language. The primary advantage of GLAMMAR as a compiler compiler is its structural simplicity. There is no need for the user to make double, losely connected, specifications, nor to program in the traditional imperative style.

GLAMMAR is perhaps better being defined as *syntax-directed text to text translation* tool or a pattern transformation language. Usage is not limited to the development of (front ends for) compilers.

2. EAG—a language description language

EAG is an acronym of Extended Affix Grammars and forms the basis of GLAMMAR. These grammars are extensions of context free grammars. We assume the reader has at least some basic knowledge of context free grammars. The extensions are:

(1) Affix expressions labeling nonterminals. This allows nonterminals to yield results. The results are composed of results of the members of a nonterminal together with constants and metanonterminals. Affixes can also be used for affix directed parsing which make it possible to describe context sensitive languages.
(2) A metalevel. The metalevel acts as a type-system and abstraction mechanism. The GLAMMAR lattice feature can be used as a very efficient unification mechanism.

3. Glammar and EAG

GLAMMAR will differ from the official implementation of EAG (when ready), Programmar, in that left recursion is not allowed and that delayed affix evaluation is not implemented. But the main difference is that GLAMMAR is designed to be a practical useful tool. Lots of the extensions of GLAMMAR have been suggested by people using prereleases.

4. Should you use Glammar

GLAMMAR is most generally defined as a pattern transformation language on text. Patterns and their transformations are specified in one uniform descriptive language. A grammar defining a pattern need not to be regular, LL or LR. It may even be ambiguous. For simple grammars a parser created by GLAMMAR is about as efficient as a YACC generated parser. More complex ones usually have more C code in their YACC implementation and therefore gain some speed in comparison with GLAMMAR .

The best comparison is to PROLOG's Definite Clause Grammars. Some differences are that GLAMMAR

- uses flow symbols
- has an automatic memorizing option (absolutely necessary for linguistic grammars)
- offers finite lattices as a metagrammar (with some well defined operators) for very fast unification and finally
- treats backtracking quite differently.

GLAMMAR is used extensively by people designing and testing grammars for English and American. GLAMMAR is also used in several Esprit projects. It is used to implement a parser for Comma (COMputable MAthematics), a syntax checker annex pretty printer for the real time description language RDSL and has been used to implement Glass (General LAnguage for System Semantics) and a prototype for an *Update Plan* compiler.

5. A brief history of Glammar

GLAMMAR is related to the products designed by H. Meijer as a start of his Ph.D. Thesis (see[29]) to make and describe an implementation of Extended Affix Grammars. The notation resembles Two level Van Wijngaarden grammars. The first prototype ("Red") was implemented in 1978. The final prototype ("Blue") in 1986. "Programmar" — the title of this thesis and the name of the designed language — contains a formal description in terms of a special designed machine language called *Update Plans*. This description is implemented without optimizations (as source to source transformations) which benefit EAG implementations more than average (what ever that means). All these products run on a IBM-370. They were implemented as a transnicer (one pass) from restricted EAG to IBM-370 assembly. When I finished my masters thesis (A compiler for Modula-2 in Blue) in 1988 it was decided to port Blue to Unix. This ported version (written in C, producing C) became available in 1989. Since then it has been extended continuously.

CHAPTER II

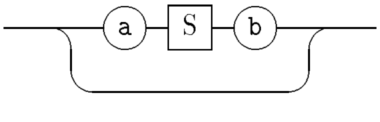
Glammar language report

This chapter describes the plain language. You will find some knowledge of context free grammars helpful. Explanation is done by introducing five levels. The first step describes recognizers for context free grammars. When a recognizer is executed it checks whether the input sentence given to it belongs to the language described by the grammar used for generating the recognizer. If the input sentence belongs to the language nothing happens — the result is void. Otherwise the result is a fragment of the input containing a line with the erroneous input position marked. In order to produce some output when a parse terminated successfully the second step introduces the most important feature of affix grammars — displays. Displays specify affix positions (arguments). Each affix position specifies either some result (as a string of characters) of a part of the grammar (a nonterminal), or it specifies how it will be composed — by concatenating terms — from other affixes of the production. The next step involves equality tests on affixes. If such an equality test fails the parsing should not proceed beyond that point. This is called affix directed parsing. Finally an affix can be not only tested on equality, but can also be the *input* of a context free grammar, or even the input of an affix grammar. But for now we will start at the beginning — the parsing of sentences described by context free grammars (CFG).

1. Level 1: Parsers described by CFGs

Let us begin with an example. Suppose we want to recognize elements from the following set { e, ab , aabb, aaabbb, $\cdots$}, here e is the empty string. Such a set is called a language and the elements are called sentences. The following picture uses syntax diagrams and defines the language L:

S



Mathematicians prefer not to draw pictures and would have defined this language as:

> The language $L = \{a^n b^n : n \geq 0\}$ is generated by the context free grammar $G = (\{a, b, S\}, \{a, b\}, R, S)$, where R contains the two rules $S \rightarrow aSb$ and $S \rightarrow e$.

In words this would read: Grammar G contains the symbols a, b, S where a, b are the terminals, R is the set of production rules and S is the start symbol. Such specifications are very much the same as the ones GLAMMAR wants you to write. However there are a few syntactic changes to the notation above. For example:

```
#Grammar producing a^nb^n
S: "a", S , "b" ; .

#The same grammar with a repeating left hand side
 T: "a", T , "b" ;
 T: .
```

Both grammars are correct GLAMMAR specifications and accept L. The general grammar in GLAMMAR does not consist of a set of productions but a set of hyperrules. A hyperrule contains all the productions with the same nonterminal on the left-hand side. Nonterminals are identifiers that may contain spaces and the symbols `*+`. Two valid nonterminal names are for instance `sequence` and `yet more elements sequence option`. Spaces as part of a nonterminal are not significant (i.e: `long name` and `longname` are the same name for the same nonterminal). The left-hand side of the first hyperrule encountered is assumed to be the start rule. Note that the nonterminal name in the left hand side of the first production belonging to a hyperrule is printed bold. In the running text however, we will refer to the names in examples by printing them in `typewriter type`

Each hyperrule is terminated with a period and consists of one or more productions. A production has a *left-hand side* (`S:`) followed by an *optional right-hand side*. A successive production may omit the left-hand side. Productions are separated by semicolons. A right-hand side consists of a number of members. A member is either a nonterminal or a terminal (sequence of characters enclosed between double quotes). Members are separated by comma's. Not all grammars obeying this syntax are accepted. For the description of context free grammars the following context conditions must hold:

- The left-hand side identifiers of productions in one hyperrule must not differ.
- The left-hand side identifiers of productions in different hyperrules must be different.
- Each nonterminal used in the right-hand side of a production must be used as the left-hand side identifier of some production or else be a builtin.

1.1. Left recursion. If a grammar conforms the requirements mentioned above an executable program (a *parser*) can be generated. However, a parser generated by GLAMMAR parses its input top down. Therefore left recursive grammars are a potential danger of nontermination or, at least, abnormal termination. For context free grammars this is quite easy to solve by transforming the grammar following the heuristic rule:

> Let $A \to A\alpha_1, \ldots, A \to A\alpha_n$ and $A \to \beta_1, \ldots, A \to \beta_m$ be all rules with A on the left-hand side, where the β's do not start with an A , $n > 0$ and there are no rules of the form $A \to A$. Then replace these rules by $A \to \beta_1 A', \ldots, A \to \beta_m A', A' \to \alpha_1 A', \ldots, A' \to \alpha_n A'$ and $A' \to e$, where A' is a new nonterminal.

As an example we will remove left recursion from the grammar accepting expressions.

 # A left recursive grammar.
 # Valid expressions are for example "x1" and "x1(x1+x2)"*
E : E, "+", T; T.
T : T, "*", F; F.
F : "(", E, ")"; "x", N.
N : "1"; "2".

#Left recursion removed :
 # E' and T' can be read as "expression tail" and "term tail" respectively.
E : T, E'.
E' : "+", T, E';.
T : F, T'.
T' : "*", F, T'; .
F : "(", E, ")" ; "x", N.
N : "1"; "2".

1.2. Ambiguities. Apart from the exclusion of left recursion there are no limitations to the grammar for which a parser can be built. In fact one of the features of GLAMMAR is the capability to handle some forms of *ambiguities*. A context free grammar G is called ambiguous if there exists a string $w \in L(G)$ with two distinct derivations in G. Ambiguity is of course necessary to be able to specify an *inherently* ambiguous language — such as any natural language. To illustrate, in a short example, the (artificial) use of ambiguous grammars for non-linguistics problems let us try to compute the Fibonacci sequence. Recall that this is the sequence $[1, 1, 2, 3, 5, 8, \cdots]$. The next number in this sequence would be 13 — the sum of the previous two numbers in the sequence. The Fibonacci number f_n of a natural number n is the nth number in the sequence. For instance f_4 is 3.

We will try to compute the Fibonacci number by constructing a grammar that has as many different left-derivations (solutions) as the Fibonacci number. Numbers are in unary notation with 1 represented as a, 2 represented as aa, etc.

A well known recursive equation for f_n is $f_1 = 1; f_2 = 2; f_n = f_{n-1} + f_{n-2}$. A first solution could be:

F : "a";

```
==>  F1  "aa"   ==>  F3  "aa"                  ==>  F4  "a"
<--  F1           ==>  F1  "a"                    ==>  F1  ""
==>  F2  "a"      <--  F1                           ==>  derivation 3 found
<--  F2           ==>  F2  ""                     <++  F1
                    ==>  derivation 1 found       ==>  F2  <--
                  <++  F2                         ==>  F3  ""
                  ==>  F3  "a"                        ==>  F1 <--
                      ==>  F1  ""                     ==>  F2 <--
                        ==>  derivation 2 found       ==>  F3 <--
                      <++  F1                         ==>  F4 <--
                      ==>  F2  <--                    ==>  F1
                      ==>  F3  ""                 <--  F3
                          ==>  F1 <--             ==>  F4  <--
                          ==>  F2 <--           <++  F4
                          ==>  F3 <--
                          ==>  F4 <--
                      <--  F3
                      ==>  F4  <--
                      <++  F3
                      ==>  F4  ""
                      ==>  F1 <--
                      ==>  F2 <--
                      ==>  F3 <--
                      ==>  F4 <--
                  <--  F4
               <++  F3
```

FIGURE II.1. Execution trace for the Fibonacci grammar

F : "aa";
F : "a", F;
F : "aa", F.

We will use this grammar to illustrate how a generated parser works. The above grammar computes f_{n+1}. To see this we discuss the execution trace of "aaa" presented in figure II.1. Note that we have marked the productions of the grammar with `F1`, `F2`, `F3`, `F4`. In the beginning there is no input read and we have the choice of trying four production. The computation in glammar proceeds depth first. Hence we enter the first production but mark the others as being "open" with no input read.

We push these open productions on the "backtrack stack". The computation starts trying production `F1` reading one input symbol leaving two symbols (this is annotated as `==> F1 "aa"`). Since not all symbols have been read at this point and the computation cannot be continued this computation fails and is closed (`<-- F1`). We have to backtrack to the find the first open production. This is `F2` with no input read. The computation continues trying production `F2` reading two input symbols but leaving one (`==> F2 "a"`). Again this computation cannot be continued and is closed (`<-- F2`).

The next open production is `F3` with no input read. This step reads one symbol, leaving two, but trying to recognize a new `F` — put on the continuation stack — with one input symbol read (`==> F3 "aa"`). Actually, before trying to recognize the symbol in the input the `F` has already been pushed on the continuation stack. After the recognition of the terminal the computation checks the continuation stack to verify that the computation has found a parse. Only if this stack is empty and all the input is read a derivation has been found and can be reported. In order to resume the computation after the recognition of the input symbol the next step (or rule) is popped from the continuation stack. Since it pushed the `F` with one symbol read just before, it will find this to be the next step. Trying this step adds 4 productions with one symbol read to the backtrack stack. The first of these (`==> F1 "a"`) is tried but is closed immediately because there was one input symbol left to read but no continuation. The second production with one input symbol read reads two input symbols (`==> F2 ""`). All symbols are read and there is no continuation. This means a derivation has been found and can be reported. After this recognition the successful production is closed and the backtrack stack is investigated to see if there exists an open path. The one found is `F3` with one input symbol read, and computation is continued.

The computation finds all solutions because it searches through all possible paths. It uses a continuation stack. This stack holds unfinished steps — things to do to complete the current path — and a backtrack stack which holds all other paths to try. If we investigate the stacks at different stages we see that at the beginning of the computation both stacks are empty. When a derivation has been found the continuation stack is empty again (otherwise the computation for that path had not been finished), the backtrack stack has approximately reached its maximum size, because the further the computation proceeds, the more alternate paths must be remembered. It cannot be established when the continuation stack reaches its maximum. It is likely to be the point at which recursion is deepest. Left recursive grammars cause a continuation stack overflow.

It is not very important to understand the working of the computation with respect to the usage of stacks. All you need to remember is that all reachable productions are entered depth first, left to right.

1.3. Cutting hyperrules. The previous grammar does not compute the Fibonacci number in terms of the number of recognitions; only a close variant of the Fibonacci number which turns out to be the next Fibonacci number. Before presenting the correct solution we will first discuss why the first attempt generates too many solutions. The problem of the previous grammar is that the corresponding grammar rule for fib 2 (F_2) generates two solutions. (F_2 and F_3F_1). Thus if F_3 or F_4 is recognized we do not want to let F_1 and F_2 succeed. This can be expressed in GLAMMAR with the *cut* (`->`) operator[1]. Here is the grammar generating the correct Fibonacci number.

```
f : "aa", f;
    "a", f, —>;
    "aa";
    "a".
```

[1]`cut` and `->` can be freely interchanged

A disadvantage of the cut operator is clearly that the order in which productions appear becomes significant. Strictly speaking, grammars containing the cut operator are not declarative. Personally I do not discourage the use of the cut operator, since it can be removed by adding composite rules to the productions following a cut operator, keeping the order of productions of the transformed grammar insignificant. In my opinion this limited cut — it is called a *slash* in Prolog — is syntactic sugar.

1.4. Look-ahead. Another solution (not using the cut operator) to the problem mentioned above uses a look-ahead. In GLAMMAR a member can be preceded by a `?`, the look-ahead operator. In contrast to the cut operator the look-ahead operator is syntactically a valid operator and not a shorthand for some predefined function. If a member x is preceded by a `?`, *after* the recognition of x, the input pointer is restored to the value it had just before recognizing x. The Fibonacci grammar can be written without the cut as shown here.

```
f :
      "aa", f;
  ?"aaa", "a", f;
      "aa";
      "a".
```

1.5. Semi-terminals. Besides terminals, we can recognize sets of terminal characters by placing them between exclamation signs (`!`). Let us present a short overview of terminals and semi-terminals.

`"a"`	the terminal letter a.
`"string"`	a terminal string. (shorthand for `"s", "t", "r" , "i", "n", "g"`).
`!abcd!`	a choice of one character from the set a, b, c, d. a shorthand for `"a"; "b"; "c"; "d"`.
`!abcd!*`	reads the longest terminal (length 0 or more) containing characters from the set $\{a, b, c, d\}$.
`!abcd!+`	reads the longest terminal (length 1 or more) containing characters from the set $\{a, b, c, d\}$.
`^!abcd!`	a choice of one character not in the set $\{a, b, c, d\}$.
`^!abcd!*`	reads the longest terminal (length 0 or more) not containing characters from the set $\{a, b, c, d\}$.
`^!abcd!+`	reads the longest terminal (length 1 or more) not containing characters from the set $\{a, b, c, d\}$.

Normally a character denotes itself. There are a few exceptions however. A `\` followed by one of `n, f, r, t \` denote a new line, form feed, return, tab and back slash respectively. If a back slash is followed by an octal number of three or less digits it denotes the character with that ascii code. In sets a `\!` denotes a `!`, in strings a `\"` denotes a `"`. In all other cases a back slash denotes itself — it need not always be escaped!

1.6. Left factoring. Left factoring a grammar is the act of deleting common prefixes between productions of some hyperrule. Due to the described evaluation order, it can be necessary in order to avoid exponential behavior. The following example illustrates this.

```
# The number of computation steps is about 2^n (where n equals the number
#           of input symbols).
# On input "aaaaaaaaaaaabbbbbbbbbbbb" this requires over 1 million computation
#  steps

# Grammar to be left factorized.
S: "a",S,"b";
  "a",S,"c";
  .

# Grammar after left factorization:

T: "a", T, TT; .

TT:"b";
  "c".

# On input "aaaaaaaaaaaaabbbbbbbbbbbbb" this requires less then 100
#  computation steps.
```

1.7. Memorization. We have seen that a simple transformation can avoid exponential behavior. But this statement cannot be reversed. Exponential behavior cannot be handled by means of some simple transformation on the grammar. Though it is possible to parse any sentence of any grammar in $O(n^{2.82})$ (where n denotes the input length) with a tabular method, the recursive backup, top down method in some cases needs exponential time. Especially grammars describing (parts of) natural languages seem to suffer from this problem. A heuristics called *memorization* is used to reduce this problem (at least in many practical situations). Memorization is based on remembering all productions that failed at a particular input position. This can prevent the backup parser from trying productions for which it is known in advance (by looking up in a table) that the production can never succeed at that particular input position.

As stated earlier the memorization heuristic can gain exponential time. The main drawback of memorization is that it requires $n \times p$ bits where n is the input length and p is the number of productions that can be memorized. In a context free grammar all productions can safely be memorized, however in an affix grammar only those production that do not have inherited affixes[2]. (or, more precisely, do not use inherited affixes for affix directed parsing) Here is a grammar that benefits from memorization.

```
#A grammar with exponential behaviour without memorizing
```

[2]We do not consider more advanced memorization strategies

```
# and linear behavior with memorizing.
S: "a",S,"b";
  "aa",S,"c";
  .

# Trying to left factorize the grammar yields  a grammar
# with exactly the same complexity.
T: "a",TT;.

TT: T,"b";
  "a",T,"c".
```

We will leave it as an exercise to see why the complexity of this grammar is polynomial when using the memorizing heuristic.

Despite left factoring a parser and memorization, a grammar can still have exponential complexity — for instance this is true for our Fibonacci grammar. In most practical cases this is not much of a problem. Compared to other well known parsing methods in which all grammars are of polynomial time and space complexity GLAMMAR needs only linear space [9]. This polynomial time behavior holds for parsers which only recognize or reject input. If for each derivation a result, or an answer, or parse tree has to be reported then a parser can always have exponential complexity — for any parsing algorithm — since there can be an exponential number of derivations.

More information about the complexity of top down, backtracking parsers can be found in [26].

2. Level 2: Transducers

Up to now we described only context free recognizers. The first extension could be in producing an answer — `Y`, for example — for each derivation. In the Fibonacci grammar — yet another version — we could implement this as follows:

```
# This fibonacci grammar produces a "Y" for each derivation,
# i.e: On the input 12 "a"'s the result is  144 a "Y"'s.
# We have introduced a new start rule.
fib ("Y">): f.

f : "a";
f : "a",  f;
f : "aa", f.

# The same grammar without a new start rule.

g ("Y">) : "a";
g ("Y">) : "a",  g (ignore>);
g ("Y">) : "aa", g (ignore>).
```

A display (the part within brackets) contains *affix positions* separated by commas. Displays are placed between parentheses and can occur anywhere within a nonterminal name or after a semi-terminal. An affix is *inherited* if a greater than (>) character precedes its position and *derived* if that symbol follows its position. A simple occurrence of an affix is just a name. In general it is an *affix expression*. An affix expression is a sequence of *affix terms* separated by the concatenation symbol `+` or the composition symbol `*`. A term is a name or a terminal. The following are examples of displays:

n (>inherited affix pos 1, `"derived position of one term"`>) *#display*
:.

(>i+i+i) **divided by three is** (i>) :.

(>i) **concatenated with** (>j) **is** (i+j>) :.

Internally all positions from displays belonging to a nonterminal are collected and stored in a list. Hence only the order in which affix positions occur is significant. The following two productions are equal.

(>i) **concatenated with** (>j) **is** (i+j>) :.
concatenated with is (>i, >j, i+j>) :.

A derived position can be thought of as the result of a function with strings as domain and range. This function takes other affix positions (inherited as well as derived) as arguments and its body consists of the concatenation operator applied to arguments and constants. When the value of a position is required (for example in the final result) its function is evaluated.

Besides distinguishing inherited and derived positions we can make another, more important, classification — between *applying* and *defining* positions. Defining affix positions are the derived ones in the right-hand side together with the inherited positions in the left-hand side. Other affix positions are applying affix positions.

A defining position must consist of an affix name or underscore; no expressions are allowed (including affix terminals). If a defining position consists of a underscore it is taken to be a don't care term — it can be substituted by a new and unique affix variable. The following context conditions must hold:

- Each occurrence of a hypersymbol must have the same number of derived positions and the same number of inherited positions in its display, if any, and the order in which the positions occur is also significant. The display belonging to the start rule must have one derived position or none at all.
- For each identifier in an expression in a hyperrule there must be a defining occurrence. I.e., that identifier must have a defining position in the production or else be the left hand side of some metarule.

An affix grammar for which the following context condition holds is called a *transducer*.

> A defining position is always a name and no other defining occurrences apply for that name. The scope of defining names in a hyperproduction does not extend over the definition of that production.

A transducer recognizes the same language as its *head grammar* — the grammar with the displays removed.

2.1. Producing parse trees. In this example we will deal with automatic generation of parse trees from context free grammars. The following simplified grammar for context free grammars will be used.

```
grammar : rules.

rules : rule, rules;
rules : .

rule : productions, ".".

productions : name, ":", members, more productions.

more productions : ";", productions;
more productions : .

members : member, more members;
members : .

more members : ",", member, more members;
more members : .

# Leave the rest, to keep it small
# member : terminal ; identifier.
# etc.
```

We can extend a context free grammar to produce a parse tree by adding — for each production — an affix position. For example:

```
p : m1, m2, m3.

# adding parse tree affix

p ("p(" +m1 +m2 +m3 +")">): m1 (m1>), m2 (m2>), m3 (m3>).
```

If we perform this transformation to the grammar describing context free grammars the result will be:

```
grammar ("grammar("+ rules+ ")\n ">): rules (rules>).

rules ("rules("+ rule+ rules+ ")\n ">):
   rule (rule>), rules (rules>);
rules ("rules()\n">): .
```

rule ("`rule(`"+ productions+ "`.)`">):
  productions (productions>), ".".

productions ("`productions(`"+ name+ "`:`"+ members+ more productions+ "`)`">):
  name (name>), ":", members (members>), more productions (more productions>).

more productions ("`more productions(;`"+ productions+ "`)\n`">):
  ";", productions (productions>);
more productions ("`more productions()\n`">): .

members ("`members(`"+ member+ more members+ "`)\n`">):
  member (member>), more members (more members>);
members ("`members()\n`">): .

more members ("`more members(,`"+ member+ more members+ "`)\n`">):
  ",", member (member>), more members (more members>);
more members ("`more members()\n`">): .

%
Example input :
  A:"a";A:"a",A,"a";A:.
Example output (a little beautified):
%

```
grammar(rules(rule(
   productions(name(A):(members(member("a")more members())
    more productions(;productions(name(A):members(member("a")
      more members(,member(name(A))
       more members(,member("a")
         more members())))
     more productions(;productions(name(A):members()
      more productions())))))).)
  rules()))
```

Since this transformation is very straightforward we can easely do this automatically. The following grammar describes this transformation.

grammar (rules>): rules (rules>).

rules (rule+rules>):
  rule (rule>),
  rules (rules>);
rules (empty>):.

rule (productions+".">):
  productions (productions>), ".".

```
productions (name+"("+quote+name+"("+quote+"+"+terms+"+"+quote+")"
              +quote+">):"+ members+more productions>):
   name (name>),
   ":",
   members (terms>,members>),
   more productions (more productions>).

more productions (";"+productions>):
   ";",
   productions (productions>);
more productions (empty>):      .

members (term+terms>, member+more members>):
   member (term>,member>),
   more members (terms>, more members>);
members ("empty">, empty>):      .

more members ("+"+term+more terms>, ","+member+more members>):
   ",",
   member (term>,member>),
   more members (more terms>,more members>);
more members (empty>,empty>):      .

%
Example input :
   A:"a";A:"a",A,"a";A:.
Example output :
   A("A("+"a"+")">):"a";A("A("+"a"+A+"a"+")">):"a",A(A>),"a";
   A("A("+empty+")">):.
%
```

2.2. Example 1: Transducer inversion. Transducer inversion is — as we will see — another example of a simple transformation on affix grammars. The grammar given just below generates a list of words from a number. It is likely that at least one of these words can be remembered more easily than the original number. This can for instance be useful in remembering someone's telephone number.

```
# "nlcr" (newline and carriage return)  and "empty" are predefined metanotions
teleword (w+nlcr>):
  number (w>).

number (x+y>):
   digit (x>), —>,
   number (y>);
number (empty>):.

digit ("0">) : "0";
digit ("1">) : "1";
```

```
digit ("a">) : "2"; digit ("b">) : "2"; digit ("c">) : "2";
digit ("d">) : "3"; digit ("e">) : "3"; digit ("f">) : "3";
digit ("g">) : "4"; digit ("h">) : "4"; digit ("i">) : "4";
digit ("j">) : "5"; digit ("k">) : "5"; digit ("l">) : "5";
digit ("m">) : "6"; digit ("n">) : "6"; digit ("o">) : "6";
digit ("p">) : "7"; digit ("r">) : "7"; digit ("s">) : "7";
digit ("t">) : "8"; digit ("u">) : "8"; digit ("v">) : "8";
digit ("w">) : "9"; digit ("y">) : "9"; digit ("z">) : "9".
```

The digits `0` and `1` are left unchanged. For each other digit there are 3 possible letters, each should be generated. For example on input `43` the words `gd ge gf hd he hf id ie if` will be listed.

The inverse grammar is given below.

```
telenum (w+nlcr>):
  word (w>).

word (x+y>):
   letter (x>), —>,
    word (y>);
word (empty>):.

letter ("0">) : "0";
letter ("1">) : "1";
letter ("2">) : "a"; "b"; "c";
letter ("3">) : "d"; "e"; "f";
letter ("4">) : "g"; "h"; "i";
letter ("5">) : "j"; "k"; "l";
letter ("6">) : "m"; "n"; "o";
letter ("7">) : "p"; "r"; "s";
letter ("8">) : "t"; "u"; "v";
letter ("9">) : "w"; "y"; "z".
```

This inverse grammar could very well be generated automatically. In fact program-inversion is a well known technique. A more detailed discussion and applications of grammar inversion can be found in [45, 44].

3. Level 3: Affix directed parsing

In transducers we required defining affix names to be unique. We will now drop this restriction. Affix grammars without this restriction are called *transnicers*.

3.1. Consistent substitution. An interesting case is when defining positions have the same name. If this is the case then the rule of *consistent substitution* is applied. This rule expresses that same names (within a rule) must have the same values. If such an equality test fail the parsing will not proceed beyond that point. Using affixes to influence the parsing process is called *affix directed parsing*. Applying the consistent substitution rule we can also describe languages that are not context free. This was not possible with transducers since a transducer recognizes the same language as its *head grammar* — which is context

```
1)
  a n b n:                    -->   a n b n:
    a sequence (n>),                   a sequence (n>),
    b sequence (n>).                   b sequence (m>),
                                       equal (>n,>m).
2)
 s (>a,>a):.                  -->  s (>a,>b):
                                       equal (>a,>b).

3)
 s (>a):                      -->  s (>a):
   b (a>).                             b (b>),
                                       equal (>a,>b).
4)
 s (>a):                      -->  s (>a):
   b (a>).                             b (b>),
   b (a>).                             b (c>),
                                       equal (>a,>b).
                                       equal (>a,>c).
```

FIGURE II.2. The consistent substitution rule factored out

free. For instance, the language $L = \{a^n b^n c^n | n \geq 0\}$ is parsed by the following grammar:

abc :
    a sequence with size (n>),
    b sequence with size (n>),
    c sequence with size (n>).

a sequence with size ("">):;
a sequence with size ("i"+n>):
    "a", a sequence with size (n>).

b sequence with size ("">):;
b sequence with size ("i"+n>):
    "b", b sequence with size (n>).

c sequence with size ("">):;
c sequence with size ("i"+n>):
    "c", c sequence with size (n>).

The consistent substitution rule can be eliminated by means of transformations, assuming there is a builtin predicate `equal` that takes two affixes only and succeeds if the two affixes have the same value. A *non-predicate* is a hyperrule that may advance the input pointer. A *predicate* therefore never increases the input pointer. Figure II.2 contains examples of the elimination of the consistent substitution rule.

3.2. Level 4: Parsing affix values. In the previous section we dropped the restriction for transducers that defining affix positions must be unique. We now also drop the restrictions that defining affix positions should be names. We allow them to be expressions as well. By dropping this restriction we are able to write even more powerful grammars. Below we present a grammar that computes the *nfib* number for GLAMMAR in a more traditional way than using an ambiguous grammar.

```
nfib (nfib>):
    !i!* (i>), nfib (>i,nfib>).

nfib' (>"","i">):;
nfib' (>"i","i">):;
nfib' (>"ii"+i,"i"+n1+n2>):
     nfib' (>i,n1>),
     nfib' (>i+"i",n2>).
```

The first member `!i!* (i>)` recognizes the input and passes it to the predicate `nfib` The grammar uses a unary base for the representation of natural numbers. In this representation concatenation means plus and splitting means subtract. Note that splits are nondeterministic. For instance a defining expression `a+b` where `a` and `b` are not metadefined, assigns to `a` and `b` $n+1$ times a different value (where n is the length of `a+b`). The semantics of defining affix expressions can be described by means of transformation. The grammar below is the transformed grammar of the previous presented grammar.

```
nfib (nfib>):
    !i!* (i>),
    nfib' (>i,nfib>).

nfib' (>E1,"i">):
    equal (>E1,>"i");
nfib' (>E2,"i">):
    equal (>E2,>"ii");
nfib' (>GG1,"i"+n1+n2>):
    set inputptr to (>GG1,G1>),
    "ii",
    ^!!* (i>),
    end of sentence,
    reset inputptr (>G1),
    nfib' (>i,n1>),
    nfib' (>i+"i",n2>).
```

Here `set inputptr to` sets the current input to the result of the evaluation of the inherited affix; it saves the old input pointer in the derived affix `G1`, `end of sentence` is a predicate that succeeds if the input is read, `reset inputptr` restores the old input pointer. The actual implementation joins — for efficiency reasons — some of these together.

4. Metalevel

As well as hyperrules, which are used to specify the head grammar, there are rules for restricting affix values. These are called metarules and the left hand side of a metarule is called a metanotion. The most simple (and most frequently used) usage of a metarule is that of an abstraction of an affix constant. Thus instead of using an affix constant you can declare a metarule for this constant and then use the metanotion for denoting the affix value. For instance the predefined metanotion `empty` could be given in various ways, for example:

empty1 :: "".
empty2 :: .
empty3 :: ""+ "" + "" + "".

All definitions for empty are equal, moreover since metarules are evaluated at compile time whenever this is possible, `empty2` is as good and efficient as `empty1`

5. Level 5: Affix grammars over affixes

So far nothing exciting about the metalevel. This changes a little with the introduction of super affixes. An example of a super affix rule is

#Definition
Filter :: @ hyper.

hyper ("substitute">):
"recognize".

#Application
app (Filter>):
application (Filter>). *# A filter*

#Is equivalent with
app (filter>):
application (out>),
hyper' (>out, filter>).

hyper' (>"recognize", "substitute">):.

The input `recognize` that is recognized by `hyper` will be the value of the defining affix position (`Filter`) of `application`. The produced value (on the derived affix position) by `hyper'`. will be used as the substituted value of `Filter` in the derived affix position in the left hand side of `app`. Note that the value produced is in general different from the value recognized. A restriction in using super affixes is that there must be at least one defining occurrence of a super affix in the production (if any). The difference between recognized and produced value seems to be in contradiction at the consistent substitution rule. However, it is checked whether the substitutes for multiply defined super affixes are identical. Therefore consistent substitution can still be guaranteed. We will try to explain this with an example which should be read carefully.

Consider the problem of selecting the longest suffixes without a common prefix of two strings. For instance, the longest suffixes of `abcdfg` and `abcdefg` are `fg` and `efg`. In fact this describes the working of the predefined predicate `compare`:

```
compare (>S,>T,rest S>, rest T>):.
```

We will try to make the same predicate without using this predefined predicate. A first solution can be:

```
compare eag (>prefix a+suffix i,>prefix a+suffix j, suffix i>, suffix j>):.
```

But this predicate generates not only the longest suffix but also all shorter ones (including empty). The following grammar is a correct solution:

```
t ("s1 = "+s1 + "\ns2 =  " + s2>):
  compare eag (> "abcdfg",>"abcdefg", s1>, s2>).

compare eag (>"",>"","">,"">): —>;
compare eag (>L + x,>L + y , i>, j>):
       compare eag (>x ,>y ,i>,j>), —>;
compare eag (>x,>y,x>,y>): .

L ::@  one char .

one char (c>): ^!! (c>).
```

As an example of how the consistent substitution rule cooperates with super affixes, consider the following grammar:

```
cs check (out>):
   cs (>"A",>"B",out>).

cs (>CS, >CS, CS>):.

CS :: @ metacs.

metacs ("Only One Substitution Value">):
   "A";
metacs ("Only One Substitution Value">):
   "B".
```

This grammar will produce as a result `Only One Substitution Value`, although the rule `cs` is subjected to the consistent substitution rule and the defining affixes of `CS` recognize different values (`A,B`), the produced values are same. For all other affixes (i.e. the ones not defined as super affixes) the produced value is the same as the recognized value. The full semantics are perhaps best expressed by the transformed grammar:

```
cs check (out>):
   cs (>"A",>"B",out>).
```

```
cs (>GG1,>GG2,CS>):
   setinputptrto (>GG2,G2>),
   metacs (CS>),
   resetinputptr (>G2),
   setinputptrto (>GG1,G1>),
   metacs (E1>),
   equal (>E1,>CS),
   resetinputptr (>G1).

metacs ("Only One Substitution Value">):
   "A";
metacs ("Only One Substitution Value">):
   "B".
```

Note that super affixes have been deleted and do not occur in the basic language anymore.

6. Composition

The concatenation operator is used for concatenation of affix values at applying positions and for splitting affix values at defining affix positions. Splitting affix values is — in contrast to concatenation — not always deterministic. For instance in `split (>i+j)`, where the position is defining, the position splits $n+1$ times if the length of the value given to the position is n.
In many cases we want to have a more deterministic way to split affix values. This can be done by using the *composition* (`*`) instead of the concatenation operator. At applying occurrences the composition operator pairs two affixes together and gives this pair a unique name. At defining occurrences the composition operator tries to unpair the position in two parts assigning the parts to the operands. The composition operator is often used to make data structures. The composition operator binds to the right (i.e $a*b*c = a*(b*c)$). Because the composition operator is right associative, grouping is necessary to be able to create certain structures. However, GLAMMAR does not allow you to use parentheses in affix expressions. For example $> a*(x*y)*c$ is not allowed. The proposed solution is to make such structures with the builtin `where`. Although the working of the builtin predicate `where` is trivial, the its presence is important for optimization. For instance:

```
t (>a*b*c):
  wwhere (>b,x*y>).

wwhere (>x,x>):.
```

is internally transformed to:

```
t (>L1):
```

```
    unpair (>L1,a>,L2>),
    unpair (>L2,b>,c>),
    wwhere (>b,L3>),
    unpair (>L3,x>,y>).

wwhere (>x,x>):.
```

In this translation step the composition operator is removed. It is obvious that the `wwhere` has become superfluous if we substitute `b` for `L3`. This is the reason why there is a predefined `where`. The compiler tries to remove a `where` if it is possible by renaming affixes. Therefore when using `where` instead of creating our own `wwhere` the result after the transformation would have been:

```
t (>L1):
    unpair (>L1,a>,L2>),
    unpair (>L2,b>,c>),
    unpair (>b,x>,y>).
```

6.1. Mixed affix expressions. The composition operator can be used to create pairs of affix values. Suppose we want to define a data structure which is composed of unordered sequences of such pairs (tables). In implementing tables we should at least create a predicate to add a pair to a table and a predicate `lookup` that given a table and the first member (the key) of a pair gets the second member (the value). More explicitly `add to` and `lookup` can be defined as:

```
add to (>table,>pair, table+pair>):.      # pair = head*tail

lookup (>table,>key,value>):
    where (>table,x+key*value+y>).
```

Here `lookup` is nondeterministic. All the pairs are examined and `lookup` succeeds for every pair it can find with the value of `key`.

6.2. Structural equality. Another effect of the composition operator and the rule of consistent substitution is that this test may implicitly check not only strings of characters but whole data structures. A short example is presented just below.

```
test ("HAVE EQUALITY ON STRUCTS!\n">):
  (>"a"*"b") equals (>"a"*"b").

(>a) equals (>a):.
```

7. No delayed affix evaluation

Besides prohibiting left recursion, the most severe limitation of GLAMMAR (not EAG) is that it fails to delay affix evaluation. Testing affixes before they are *instantiated* will result in a fatal runtime error message report. For example the following grammar will, when compiled and run, produce the error message `runtime error: evaluating undefined affix.`

```
# Example of a grammar that will cause a runtime exception
```

```
# At compile time this will not be detected.
runtime exception:
        test (>a),
        define (a>).
test (>"a"):.
define ("a">):.
```

However the above mentioned prohibition does not imply that you can't ever use affixes before they are defined. Just be sure not to evaluate an affix that is used before being defined. An affix will be evaluated if it is subjected to the consistent rule or whenever a builtin needs to use its value (see section III. 9 for the details). For example in the following grammar the affix flow causes no problems.

```
# Use affixes before they are defined.
# Produces "aa".
use before instantiated (aa>):
    use (>a,aa>),
    instantiate (a>).

use (>a,a+a>):.
instantiate ("a">):.
```

8. Left recursion elimination in affix grammars

The left recursion prohibition was already addressed in section 1.1. In this section we will show how the left recursion elimination transformation can be extended to deal with left recursive grammars including affixes. We first show how this can be done for transducers with derived affix positions only. Furthermore we limit ourselves to immediate left recursion, and consider only grammars with one derived affix. Note that all derived affixes can be merged into one position anyway — using composition (*) and that left recursive productions can be transformed in a straightforward manner to immediate left recursion.

```
A lr (A lr+c>):
  A lr (A lr>),
  c (c>);
A lr (b>):
  b (b>).

# Transformed grammar:
A nlr (A nlr>):
  b (b>),
  A nlr tail (>b, A nlr>).
A nlr (b>):
  b (b>).

#This rule can be subjected to left–factoring
```

```
A nlr tail (>b, A nlr>):
  c (c>),
  A nlr tail (>b+c, A nlr);
A nlr tail (>b, b+c>):
  c (c>).
```

The following example deals with transducers in which inherited positions are allowed, but affix directed parsing is still excluded.

```
# Here "F x y" denotes an affix expression of affixes "x" and "y"
# "G x y" and "H x y" stand for such affix expressions also.
A lr (>ai,F ao c>):
  A lr (>G ai c, ao>),
  c (>H ai ao, c>);
A lr (>bi, bo>):
  b (>bi,bo>).

# Transformed grammar:
A nlr (>ai, ao>):
  b (>bi, bo>),
  A nlr tail (>ai, >bo, bi>, ao>).
A nlr (>bi, bo>):
  b (>bi, bo>).

A nlr tail (>ai, >lt, G aai c>, ao>):
  c (>H aai lt, c>),
  A nlr tail (>ai, >F lt c, aai>,ao>);
A nlr tail (>ai, >lt, G ai c>, F lt c>):
  c (>H ai lt, c>).
```

Finally we give an example of the transformation in which context conditions are included. The context conditions are expressed explicitly by means of predicates. Compared with the "transducer" case the only difference is the presence of these predicates.

```
# Here "F x y" denotes an affix expression of affixes "x" and "y"
# "G x y" and "H x y" stand for such affix expressions also.
A lr (>ai,F ao c>):
  A lr (>G ai c, ao>),
  c (>H ai ao, c>);
A lr (>bi, bo>):
  b (>bi,bo>).

# Transformed grammar:
A nlr (>ai, ao>):
  b (>bi, bo>),
  A nlr tail (>ai, >bo, bi>, ao>).
A nlr (>bi, bo>):
```

```
  b (>bi, bo>).

A nlr tail (>ai, >lt, G aai c>, ao>):
  c (>H aai lt, c>),
  A nlr tail (>ai, >F lt c, aai>,ao>);
A nlr tail (>ai, >lt, G ai c>, F lt c>):
  c (>H ai lt, c>).
```

9. Left factoring affix grammars

The usefulness of left factoring was already shown in section 1.6. In this section we will show how the left factoring transformation can be extended to deal with grammars with affixes as well. The following example will provide left factoring of affix grammars.

```
# Non left factored grammar
# Simple case
a (b+c>):
   B (b>),
   c(c>);
a (d+b>):
   B (b>),
   d (d>).

# Left factored grammar
a ( more a>):
   B (b>),
   more a (>b, more a>).

more a (>b, b+c>): c (c>);
more a (>b, d+b>): d (d>).

# Non left factored grammar
# More general case.
a (>a, c>):
  B (>c, b>),
  C (>a, >b, c>);
a (>a, d>):
  B (>d, b>),
  D (>a, >b, d>).

# Left factored grammar
a (>a, g>):
  B (>g, b>),
  G (>a, >b, g>).

G (>a, >b, c>):
  C (>a, >c, b>);
```

```
G (>a, d>):
  D (>a, >b, d>).
```

10. Memorizing affix grammars

we have already stressed the need of memorizing in section 1.7. However, we still have to address a few extra items related to memorization. These are caused by the introduction of determinism and affix directed parsing. The idea of memorization was to remember failures for each production and each input position. It could then be established — independent of the stage of a computation — that it is doomed to fail without actually doing the computation. But when a production uses its inherited affixes for affix directed parsing some paths are closed because of the mismatch of affixes. Hence the memorization table should also remember for each production the values of the inherited affixes since these can differ on different stages of a computation. However, this would require much more memory.

The most robust way to deal with the problem and avoiding problems with affix directed parsing is to limit memorization to productions that have no inherited affix positions. In GLAMMAR there are refinements to this strategy. This is because GLAMMAR can do computations over the grammar. It is computed which affixes are used for affix directed parsing, and only those productions that do not use their inherited affixes are memorized entirely. Furthermore GLAMMAR can memorize parts of the grammar. For example if a production with m members uses only one inherited affix for affix directed parsing and this affix position is used in its n^{th} member the part consisting of the first n members can be memorized. Also when the first k members are deterministic and $n < k < m$ then the last $m - k$ members can be memorized.

11. Natural languages

Natural language descriptions differ from any other (semi-) formal description (such as programming languages). Therefore a number of features have been added to make the GLAMMAR system a convenient tool for linguistics. These features are *memorizing, finite lattices*, and a *lexicon interface*. Since memorizing as already been discussed we skip the description at this point and continue with the lattice part.

11.1. Finite lattices. We have seen that, until now, an affix position always has a flow attached. It is either derived or inherited. This is for example one of the main differences between GLAMMAR and Prolog. Due to the ommission of flow specification Prolog has unification while GLAMMAR has some simpler test and assign mechanism. Both have their advantages. It would be nice though if in GLAMMAR we could drop the flow symbols on certain occasions. This is the case when affixes are used to specify some restriction, a value that must be in a set containing primitive unordered elements. Then it is also desirable that the consistent substitution rule does not impose that both affixes should have equal values but that both affixes are assigned new values — the intersection of both.

This implies that the domain of such affixes must be suitable to perform such operation.

In GLAMMAR you can omit the flow symbol of an affix position. This affix position is then taken to be a member of a lattice. Each such application of a lattice member has to be metadefined. Its name must occur on the left hand-side of a metarule. In order to reduce the chance of mistakes — when you just forget to type a `>`, this metarule has a little different form. After the symbol `::` you must also put a pair of curly brackets `{}`. For example:

NUMBER::{} FIRST ; SECOND; PLU.

Here `FIRST, SECOND` and `PLU` are primitive members of the lattice `NUMBER`. Strange constructs like `FIRST::{} FIRST.` are necessary to be able to use primitive members in the grammar. Also it is important that the top members of a lattice describes (indirectly) all other members of the lattice. The compiler uses this information to compute lattice groups. If, between applications and definitions, an affix position is occupied with lattice members of different groups, this is taken to be an affix type mismatch and you will be informed about it. In GLAMMAR a lattice (a set with a forbidden element $\perp$) is implemented as a power set of maximally 32 primitive unordered members. The limitation of the number of primitive members does not limit the total number of primitive members. This is because there may be any number of lattices.

11.2. Transformations of lattice affixes. A lattice is used to describe some context condition. It is a bit sequence representing valid solutions. The relation between affixes represented by character strings is not unique. There are two ways to convert lattices to strings. We will explain this by presenting an example that recognizes a number of simple English sentences. In addition we will show extend recognizers to parsers (i.e: recognizers that produce parse trees) The recognizer is defined as follows:

sentence : nounphrase (person,number,gender),
        verbphrase (person,number,gender).

nounphrase (P1,SINGLE,gender): "I ";
nounphrase (P2,number,gender): "you ";
nounphrase (P3,SINGLE,MALE): "he ";
nounphrase (P3,SINGLE,FEMALE): "she ";
nounphrase (P1,PLU,gender): "we ";
nounphrase (P3,PLU,gender): "they ".

verbphrase (P13,SINGLE,gender): "was working";
verbphrase (person,PLU,gender): "were working".

 # The first lattice (0) , first rule must describe all elements (not only
 #   the primitive ones.
 # This lattice has 3 elements (out of 32 max)
person :: {} P1; P2; P3; P13.
P1 :: {} FIRST.   *# Not defined element taken to be a primitive element*

P2 ::{} SECOND.
P3 ::{} THIRD.
P13 ::{} P1; P3.

 # This lattice (1) has 2 elements (out of 32 max)
gender ::{} MALE; FEMALE.
MALE :: {} MALE. *# MALE primitive element*
FEMALE:: {} FEMALE. *# ...*

 # This lattice (2) has 2 elements (out of 32 max)
number :: {} SINGLE; PLU.
SINGLE:: {} SINGLE. *# SINGLE primitive element*
PLU:: {} PLU. *# ...*

In the above example a recognizer is specified. We will now extend this grammar so that it will produce a parse tree (in text representation). First we compile with `glammar -rparse-tree test` and send to output to a file. This results in an equivalent grammar delivering a parse tree but with lattices not taken into account. We have inserted this lattice part manually.

sentence ("(sentence "+nounphrase+verbphrase+"["+person+number+gender+"])\n">):
nounphrase (nounphrase>,person,number,gender),
verbphrase (verbphrase>,person,number,gender).

nounphrase ("(nounphrase I)\n">,P1,SINGLE,gender):
"I ";
nounphrase ("(nounphrase you)\n">,P2,number,gender):
"you ";
nounphrase ("(nounphrase he)\n">,P3,SINGLE,MALE):
"he ";
nounphrase ("(nounphrase she)\n">,P3,SINGLE,FEMALE):
"she ";
nounphrase ("(nounphrase we)\n">,P1,PLU,gender):
"we ";
nounphrase ("(nounphrase they)\n">,P3,PLU,gender):
"they ".

verbphrase ("(verbphrase was working)\n">,P13,SINGLE,gender):
"was working";
verbphrase ("(verbphrase were working)\n">,person,PLU,gender):
"were working".

person :: {} P1; P2; P3; P13.
P1 :: {} FIRST.
P2 ::{} SECOND.
P3 ::{} THIRD.
P13 ::{} P1; P3.

gender :: {} MALE; FEMALE.
MALE:: {} MALE.
FEMALE:: {} FEMALE.

number :: {} SINGLE; PLU.
SINGLE:: {} SINGLE.
PLU:: {} PLU.

We have added the lattice affixes in the production for `sentence` by simply putting it on a derived position. The compiler adds a builtin predicate (`transform lattice`) at the end of the production and renames the lattice affixes occurring in derived or inherited positions. In fact the start of the transformed grammar is:

```
# The internal predicate "explinterset (>x,>y)" intersects the bitpattern
#  in "x" with the bitpattern "y" (after converting the decimal value to
#  binary) and fails if no bits in x remain set).
# "intersect" does the same thing except that both its arguments are already in
# the internal format.
#
# Transform lattice takes a lattice group (in decimal) and a bit pattern
# specifying a subset of all the members of the group.
# Note that this subset may not be empty.
# From this it computes the names of the members that are in the set.
# A list of all the menbers in each lattice group is constructed compile time.
sentence ("(sentence "+nounphrase+verbphrase+"["+T0+T1+T2+"])\n">):
   nounphrase (nounphrase>,person,number,gender),
   explintersect (>gender,>"3"),
   explintersect (>number,>"3"),
   explintersect (>person,>"7"),
   verbphrase (verbphrase>,E4,E5,E6),
   intersect (>E6,>gender),
   intersect (>E5,>number),
   intersect (>E4,>person),
   transformlattice (>"0",>person,T0>),
   transformlattice (>"2",>number,T1>),
   transformlattice (>"1",>gender,T2>).
```

As stated earlier there is a choice of how the conversion of lattice affixes should be done. The first possibility (default) is to represent lattices as a set $\{el_1, el_2, .., el_n\}$. On input `I was working` this produces one parse tree:

```
(sentence (nounphrase I )
          (verbphrase was working)
          [{FIRST}{SINGLE}{MALE,FEMALE}])
```

Another possibility is to take one member from the set and, upon backtracking, use another, until there are no more solutions. This is the more traditional way to transform (implement) lattices. If compiled with option `-L` this kind of behavior is default; in our example there are two parse trees, one for each solution of `GENDER`.

When transformations on lattice affixes are frequently used this way of doing is strongly dissuaded because there will be an exponential growth of the number of parse trees.

```
(sentence (nounphrase I )
       (verbphrase was working)
        [FIRST SINGLE MALE])
(sentence (nounphrase I )
          (verbphrase was working)
          [FIRST SINGLE FEMALE])
```

11.3. Lexicon interface. The lexicon interface is implemented as the maintenance of up to 32 connections to a separate program called *lexicon*. Therefore this feature only exists on unix-like machines. Chapter IV describes some details about (unexpected) properties of such a connection.

The program `lexicon` takes as argument a compiled lexicon. A lexicon can be compiled with the program `make-lexicon`. These two programs are very similar and use the standard unix database routines[3] `dbm_open`, `dbm_close`, `dbm_fetch`, and `dbm_store`. This makes the programs become very small (20 lines C each).

The lexicon program works as follows: Main operation is an endless loop in which a word is read from the program's standard input. A word should be ended by a new line. This word is used as a key in `dbm_fetch`. If the word can be found the program *lexicon* writes a one ("1") to the standard output; if not found a zero.

Below we have given a small part of a grammar for English which uses the lexicon interface.

 # SENT sentence
 # SIIM SUSI + IMPR
 # SING singular
 # SIPR SUSI + PLUR
 # SUBJ subject
 # SUBS noun; adjectives already generated
 # SUPH subject phrase
 # SUPL SUBS + PLUR
 # SUSI SUBS + SING
 # PLUR plural
 # ADJE adjective

START (t>):
 # setup three connections to the program "lexicon"

[3]The Gnu versions of these programs are preferable

setup connection to unix (>"lexicon ADIE",>LEX ADIE),
setup connection to unix (>"lexicon SIIM",>LEX SIIM),
setup connection to unix (>"lexicon SIPR",>LEX SIPR),
sentences (t>).

 # These metanotions are given a (channel) number (<32) by "setup connection to unix"
LEX ADIE::.
LEX SIPR::.
LEX SIIM::.

sentences (t+tt>):
SENT (t>), more sentences (tt>).

 # ...
 # ...
 # ...
 # ...

SIPR("SIPR("+name+")">):
identifier (name>),layout,
talk to unix (>LEX SIPR,>name+nlcr,"1">). *# look for the name in the lexicon SIPR*

SIIM("SIIM("+name+")">):
identifier (name>),layout,
talk to unix (>LEX SIIM,>name+nlcr,"1">).

ADIE("ADIE("+name+")">):
identifier (name>), layout,
talk to unix (>LEX ADIE,>name+nlcr,"1">).

12. Experimental Feature: Arbitrary Integer Arithmetic.

So far affixes have always been strings, except for lattices and composite affixes. Lattices are converted automatically to strings when used in a derived or inherited position. Composite affixes are not converted, it is the programmers responsibility to use them properly, in such a way that they do not appear in the output. For example the following erroneous phrase of a grammar writes a composite affix to the output medium: "`type out (>empty*empty)`". This would appear there as something like `"!<092993>!"`. Integer affixes can be seen as special composites, and hence are subjected to the same restrictions. Integers can only be defined as metarules. They are defined as a metaconstant number without the quotes surrounding terminals. For example:

mint 0 :: 0.
one :: 1.
big :: 1234567890123456789012345666789.

Composite affixes have the two predicates `pair` and `unpair` working on them.

These predicates are automaticly generated when removing the composition operator depending on whether the occurrence was defining or applying. Integers do not have special operators attached, but there are numerous predicates working on them. Perhaps the most important ones are those converting strings to integers and vice versa. For example the following grammar reads two numbers (as strings of course) computes the sum and prints the result plus one.

```
sum plus one ("Result:  "+out+nlcr>):
  read mint (m1>), layout,
  read mint (m2>), layout,
  int (>m1) plus int (>m2) is int(m3>),
  int (>m3) plus int (>one) is int(m4>),
  free int (>m3),
  free int (>m1),
  free int (>m2), # don't use these anymore, so make them zero
  int (>m4) is dec (out>).

read mint (m1>):
  !0123456789! * (d1>),
  dec (>d1) is int (m1>).

one :: 1.
```

See section 9 for a complete listing of the available predicates working on integers. Furthermore a small module using integer arithmetic is written to support exact numbers. Appendix B includes this implementation.

CHAPTER III

Using Glammar

At this point it is assumed you have some experience of using GLAMMAR. If this is not the case then it is suggested that you read the manual page first (page 69).

1. Including comments

Comments can be inserted anywhere, just as white space. There are two *comment modes*. The first one is the comment to end of line mode. This mode takes effect when you type a `#`. For example:

```
# Comment are started with a # char.
# On the next line there will be program text
rule:.      # a well as some comment too
# End of example.
```

The second mode is useful for larger comments. In this mode all text between percent signs (`%`) is taken to be comment.

```
%This is supposed to be a long comment
but I am too lazy to make such a thing.%
```

The pretty printer uses this form to let you specify pragmatic actions.

```
%TEX
\section{Comments}
%
```

It takes `%TEX`, `%^TEX`, `%$TEX` to be an escape to type plain TEX stuff (See section 11)

2. Data Structures

Though glammar is not designed to be a programming language it is nice to be able to store and retrieve information. Heavy use however, might (for reasons of efficiency) suggest you to use another language.

2.1. High level data structures. High level data structures can be implemented using mixed affix expressions and nondeterminism. In the following example we present a solution to the usual restrictions concerning the use of identifiers within a certain environment.

```
# Demonstration of how to implement a high level data structure.
# It describes two comma seperated lists, a definition list and a
# application list.
# It is checked that all applications are defined and no multiple
# definitions are given.
# Each definition is given an unary number.
# The output (code) wil be a sequence of pairs (x,y) where x is a
# application and y is a unary number (string of 1).
demo (code>): layout,
      definition list (env>),
      application list (>env, code>).

definition list (env>):
      def list( >empty,>empty, env>).

def list (>nr, >env, new env>):
     identifier (id>),
     not multiple defined (>env, >id),
     layout,
     rest def list (>nr+"1", >env + id * nr , new env>).

not multiple defined (>_+id*_+_, >id):  —>,
     type out (>id+ " already defined\n"), fail;
not multiple defined (>env, >id): .

rest def list (>nr, >env , new env>):
     ",", layout,
     def list (>nr, >env, new env>), —>;
rest def list (>nr, >env , env>):
    ";", layout.

application list (>env, "("+id+","+nr+")\n"+rest code>):
     identifier (id>), layout,
     (>id) is known in (>env) as (nr>),
     rest app list (>env, rest code>).

rest app list (>env , code>):
     ",", layout,
     application list (>env, code>), —>;
rest app list (>env , empty>):
    ".", layout.

(>id) is known in (>_+id*nr+_) as ( nr>): —>;
(>id) is known in (>env) as ("UNKNOWN">):.
```

2.2. Tuples and algebraic data structures. It has already been explained that tuples can be made using composition. Such tuples can be used to implement all kinds of data structures. In fact in some functional languages tuples are used to implement algebraic data structures. Specification of an algebraic data structure resembles that of a context free grammar.

2.3. Tables. Each affix can be the name of an initially empty global table. In a table you store pairs of keys and values, both arbitrary affixes. There are builtin predicates that operate on tables. These are `add to, lookup, excludes, includes, value of, contents of, select, delete, size`. For further information we refer to the description of builtins. Local tables could be implemented using composition. For example, a slightly modified `add to` and `lookup` can be defined as:

```
add to (>tab, >a, >b, pair*tab>):
    where (>a*b, pair>).


lookup (>head*tail, >a, b>):
    where (>head, a*b>);
lookup (>head* rest, >a, b>):
    lookup (>rest, >a, b>).
```

However such tables are in fact lists and therefore not very efficient. The builtin tables are implemented as a binary tree of lists of (key, value) pairs. Looking for an entry in a table is of complexity $O((\mathrm{LOG}n) + m))$ where n is the number of tables and m is the mean number of entries per table.

3. How to write deterministic grammars

A LR or LL like grammar (parser) specified in GLAMMAR can not really compete with hand- or YACC-written parsers. This is because of the inherent non-deterministic character of GLAMMAR . Even if the speed is somewhat comparable, memory consumption certainly is not.

Fortunately, GLAMMAR offers the user the opportunity to specify parts of the grammar deterministically. For those deterministic parts no backtrack space is allocated, reducing the overall memory usage for LL or LR grammars to decent proportions.

3.1. The Compilers Decision Algorithm. The strategy for deciding whether a hyper-rule is deterministic or not, unfortunately is unsophisticated. A hyper-rule is detected by GLAMMAR as deterministic only if all members in all productions are deterministic and all productions but the last production contain the cut operator. A few builtins are not deterministic but most are. Most of these non-deterministic builtins can be forced (option -d) to behave deterministically, though this can give unexpected results (see the description of builtins for more details). If option -d is used, the root rule has to be deterministic (the root rule is deterministic only if all the rules used are deterministic) .

As an example of a deterministic grammar we will present an efficient implementation of the language consisting of a sequence of "a"s (it can be generalized to an arbitrary sequence)

```
a sequence :
   a, —>, a sequence;
a sequence :
   .

a : "a".
```

This grammar will be translated without backtracking because it is recognized by the compiler as a deterministic grammar. If we had omitted the `"->"` in the first production of `a sequence` only the rule `a` would have been deterministic and, if we would add the production `a: "b"` without adding a cut to the first production of `a` no part of the grammar would be deterministic.

3.2. Forcing determinism. We have already mentioned that there is an option "-d". This is useful because certain builtins normally must undo the result of actions performed on global tables. For example `add to` must delete the item added to the table, upon backtracking. This also holds for the predicates `assign, repair, set line file,..` etc (see the description of builtins for full details).

Consider however a grammar in which only the builtins mentioned are sources of nondeterminism. Then obviously there is no need for backtracking because the undoing of actions of builtins is not meant to produce more results. This is exactly where the option "-d " is meant for. It is an error to specify the option "-d " in the case the grammar is not deterministic.

There are two builtins that cannot be forced to act deterministiccally. Both are generated by GLAMMAR during transformation steps. We will discuss the situations were they are generated. The first of these is `nestarset`. This predicate is generated when a defining affix position is occupied with an expression which contains an affix name (not metadefined) as a (all but the last) term. (e.g: `split (>x+y):.` without `x` and `y` being metadefined). This builtin can not be forced to act deterministically because the user can make deterministic splits by giving metadefinitions of the terms — in our example `x` and `y`. The following grammar makes a deterministic split of an affix-value.

```
split (X+Y>):.

X:: @ x .
Y:: @ y .

x ("x"+x>): "x", —>,x (x>);
x (empty>):.

y ("y"+y>): "y", —>,y (y>);
y (empty>):.
```

The second builtin which can not act deterministiccally is `traditional transform lattice`. This builtin generates all the elements in a lattice one by one.

4. Separate compilation

The compilation of a GLAMMAR source is split into two main phases. The first phase generates the C code for the transnicer, the second phase generates the transnicer by calling the systems C compiler and waits for this to finish. While using GLAMMAR you might have noticed that this second step is much (about 20 times) more time-consuming then the first phase. In order to reduce compilation time when changing and recompiling your grammar happens often, you can cut your grammar in pieces (files) of rules, put the names of these files in a file called the "part list" file, compile them separately and link them together into one executable. The part list file implicitly has to end with ".p", the file names in this part file are plain (i.e: there are no extensions added). The definition of a rule may not be spread over different parts. Besides the part name, each line in the part list file, contains a checksum of the part. Initially you must give this checksum the value zero; GLAMMAR will replace these with the correct values. For example the following session compiles a grammar separately:

```
>> cat grammar.p
g1 0
g2 0
g3 0
>> cat g1
expr:
 term, "+", expr;
expr:
 term.
>> cat g2
term: factor, "*", term;
term: factor.
>> cat g3
factor: !0123456789! +.
>> glammar -p grammar
gcc -c g1.c
gcc -c g2.c
gcc -c g3.c
gcc -o grammar g1.o g2.o g3.o /usr/local/lib/builtin.a -lmp
>> ls
g1 g1.o g2 g2.o g3 g3.o grammar* grammar.p
>> cat grammar.p
g1 21925022
g2 9664316
g3 3400513
>>
```

If we make the rule `more term` (in `g2`) deterministic and recompile the grammar then file `g1` will be too recompiled because the deterministic property of the rule `more term` has changed.

```
>> cat g2
term: factor, "*", term,->;
term: factor.
>> glammar -p grammar
gcc -c g1.c
gcc -c g2.c
gcc -o grammar g1.o g2.o g3.o /usr/local/lib/builtin.a -lmp
>> cat grammar.p
g1 21912767
g2 9835690
g3 3400513
```

Semantically there is no difference between cutting a grammar in arbitrary pieces, compile them separately and link them into one module, or compile the grammar in one (long) go. More specifically all hyperrules and metarules are defined external. Introducing a new metarule in one part might require some other parts to be recompiled just as changing the deterministic property of hyperrules does. Therefore the out of date method of source files to implement separate compilation (used for example in `make` is not sufficient. As a solution we use a checksum representing all semantic properties of a part. This checksum is stored in the part list file and is updated automatically.

4.1. Hiding Object files. The idea behind separate compilation is that for each part an object file is generated and stored in order to save compilation time. Since these files will have about the same (hopefully long) lifetime as your source files it is a good idea to hide them in a separate directory. In fact GLAMMAR always tries to put objects in the subdirectory `.glammar` (watch the leading period). However, if this directory doesn't exists GLAMMAR will put objects in the current directory.

5. Implementing error recovery

5.1. Error recovery. Since GLAMMAR creates recursive backup parsers, error recovery is not possible automatically. To report more than one (the default) error message one can use the builtin `error message`. This predicate generates an error message that is based upon the most advanced input position pointer ever reached, at the point the predicate was called. The following example produces the normal error message upon input failure, on a do-it-yourself basis.

new start rule (old start rule>):
  old start rule (old start rule>), end of sentence, —>;
new start rule (empty>): *# the affix empty will not be printed*
  error message (x>),
  type out (>x).

Error recovery can be implemented also, as you can see from the following example describing sequences of elements. It is done by generating a message and skipping tokens until the next token that can be used as a synchronization point. Here we assume an element is terminated by a point symbol.

elements:
  element,
  elements,—>;
elements:
  end of sentence, —>;
elements:
  error message (x>), type out (>x),
  recover.

recover:
  ^!.! *, more recover.

more recover:
  ".", layout, —>;
more recover:
  end of sentence.

yet more recover:
  ?element, elements, —>;
yet more recover:
  recover.

6. Debugging

6.1. Undesirable ambiguities. Ambiguities can be tracked down using the -a option when a parser is generated. For example:

```
>> cat ambiguous.g
test: amb.
amb:"a"; "a", amb; "aa", amb.
>> glammar -a ambiguous
line 1: (message) no left-hand side in ‘amb’
gcc -o ambiguous ambiguous_.c /usr/local/lib/glammar.a
>> cat i
aaaa
>> ambiguous i
runtime message: ‘amb’ ambiguous, 2 ’a’
runtime message: ‘amb’ ambiguous, 3 ’a’
```

6.2. Execution trace. If you have compiled your grammar with -t or -T you get an execution trace. For example:

```
>> cat trace.g
test (trace>):
```

```
   !i!* (i>), trace (>i,trace>).


trace (>"","i">):;
trace (>"i","i">):;
trace (>"ii"+i,"i"+n1+n2>):
   trace (>i,n1>),
   trace (>i+"i",n2>).
>> glammar -t trace
gcc -o trace trace_.c /usr/local/lib/glammar.a
>> cat i
iiii
>> trace i
0 ==> Init_One_Star 0 'i'
1 ==> test 0 'i'
2 ==> trace (iiii) EOF
1 3 ==> trace (ii) EOF
2 4  ==> trace () EOF
1 5 ==> trace (i) EOF
6 ==> trace (iii) EOF
1 7 ==> trace (i) EOF
8 ==> trace (ii) EOF
1 9 ==> trace () EOF
10 ==> trace (i) EOF
iiiiiiiii1 10 <++ trace EOF
1 9 <++ trace EOF
1 8 <++ trace EOF
1 7 <++ trace EOF
1 6 <++ trace EOF
1 5 <++ trace EOF
2 4  <++ trace EOF
1 3 <++ trace EOF
1 2 <++ trace EOF
1 <++ test 4,0 'i'
```

Note that only inherited affixes are traced. The number at the very beginning is the static level and the second number the dynamic level number. If your input is quite large then tracing the output this way can be a extensive task, since the error is most of the time somewhere near the end of a computation. Therefore it is possible to start the tracing at a dynamic level number. For example:

```
>> trace -t9 i
1 9 ==> trace () EOF
10 ==> trace (i) EOF
iiiiiiiii1 10 <++ trace EOF
1 9 <++ trace EOF
```

Option -T implies (at compile time) that builtins are traced also. Some of these builtins will have been inserted by the compiler.

7. Increasing stack sizes

If a stack overflow occurs you can rerun the grammar by specifying a larger stack size. There are three stacks that can cause overflow and these can be increased as follows:

- h*n* Set affix heap size to *n**512K
- c*n* Set character size to *n**512K
- c*n* Set continuation size to *n**512K

8. Line loop mode

Normally a grammar reads all its input at once and checks whether this input belongs to the grammar. In "line loop" mode this behavior is changed. A grammar running in "line loop" mode (-l option) processes one line at a time. This line is taken to be the whole sentence. When this sentence is accepted the result is written to some output medium. This feature is mainly added for efficiency reasons. For example:

```
# Grammar running in line loop mode
# This is equivalent with
#    egrep "^The"
#  though this command is about  50% faster
#
line (line>):
  ?"The", ^!!*(line>).

# Same grammar not running in line loop mode
#  is much slower for "big" input files (>>100K)

lines (line+lines>):
   line (line>),
   lines (lines>), —>;
lines (empty>):.

line (line>):
  ?"The", ^!!*(line>).
```

9. Builtins

This section contains the description of the current set of builtins. Some builtins may not be redefined. The comment statement `#plain` indicates this for hyperrules. None of the metabuiltins may be redefined. There are also some builtins that you may not even use. This is the case when a name ends with a underscore. Other properties of a builtin are implicit in the form of its definition. These properties are:

(1) deterministic builtins have only one production:
builtin: x.

(2) nondeterministic builtins that can be forced to behave deterministicly (-d option) have two productions:
builtin : x;
builtin : x.

(3) nondeterministic builtins have three productions:
builtin : x;
builtin : x;
builtin : x.

(4) builtins that always succeed have no members in the right hand side:

builtin : .

(5) builtins that do not produce empty have a terminal ("a" by default) in the right hand side:
builtin : "x".
(6) builtins that use an inherited affix for affix directed parsing have a single affix terminal on this position:
builtin (>"x") : x.
(7) builtins that do not use an inherited affix for affix directed parsing have a single affix variable on this position:
builtin (>x) : x.

empty ::.
nlcr :: "\n".
quote :: "\"".
back slash :: "\\".

These are some metanotions which you can use to recognize frequently used special symbols.

generation time :: "Fri Sep 22 11:53:37 1989".
generation date :: "Sep 22, 1989".

The time and date the grammar was generated (GLAMMAR was invoked). This metarule is set when you compile the grammar. The definitions below show their form only.

All special symbols can be denoted by giving the octal ascii representation escaped by a back slash. To remain compatible with earlier implementations some special symbols are predefined.

#plain
end of sentence= ?"\000".

Tests for reached the end of the input.

nlcr= "\n".
quote = "\"".
back slash = "\\".

Recognize a newline, quote or back slash respectively.

This is a list of predicates which have been implemented in very early versions also.

#plain
equal (>"x",>"x")= "a".
#plain
not equal (>"x",>"y")= "a".

Recognizes the empty string ('succeeds') if the values of `x` and `y` are (not) equal.

compare (>x,>y,tail x>, tail y>)= .

Let K be the longest common prefix of (the values of) `x` and `y`. Then `tail x` and `tail y` will assigned values such that `x` = K + `tail x` and `y` = K + `tail y`.

charnum (pos>)= .
settabstop (>tabstop)= .
linenum (line>)= .
length (>in,length>)= .

The affix `pos` in `charnum` is given the number of characters, in decimal, already recognized in the input. The affix `tabstop` in `settabstop` specifies the value (in decimal) of the variable `tabstop`. This value is used in `charnum`. The default value for `tabstop` is 8. The affix `line` in `linenum` is given the value (in decimal) of the current line number. The affix `length` in `length` is given the value (in decimal) of the number of characters of affix `in`.

some name (name>)= .

The affix name is given a value that is composed of 32 upper and lower case letters. The number of characters of this name is the minimum that is necessary to yield a unique name. Each name can occur only once.

GLAMMAR allows the manipulation of certain ordered sequences of pairs of values for historic reasons called *lists* although *tables* would be better instead.

add to (>table, >key,>value)= ;
add to (>table, >key,>value)= .

Each table is recorded under a name which is an arbitrary affix value (it may be empty, contain newlines, be obtained from `some name` etc.) However it may not be composite (i.e: `add to (>x*y,k, >v)` is not allowed). Initially every possible affix value is the name of an empty table. The builtin `add to` puts the pair (K,P) in front of the table named L, where L,K and P are the values of `table, key` and `value` respectively. Upon backtracking the pair is deleted from the table, unless it is forced to behave deterministic.

value of (>"table", >"key",value>)= "a".

If (K,P) is the first pair, if any, in table L such that L and K are the values of `table` and `key` respectively (which most be both known), then the value of `value` is P otherwise it is the empty string.

select (>"table", key>,value>)= "a";
select (>"table", key>,value>)= "a";
select (>"table", key>,value>)= "a".

The values of key and value are K and P respectively, where (K,P) is any pair in the table L, where L is the value of `table` (which must be known). Note that `select` succeeds once for each pair.

delete (>"table", key>,value>)= "a";
delete (>"table", key>,value>)= "a";
delete (>"table", key>,value>)= "a".

`delete` is quite a strange predicate. Basically it deletes all pairs in the table, one at a time. When a pair is deleted, upon backtracking the deleted item is not restored. Instead one more element is deleted, until the table is empty.

contents of (>"table",>pair pair sep, >key val sep, contents>)= "a".

`contents of` finds the contents of a table in one go, provided the table is not empty. The list of pairs are separated with `pair pair sep` and the two elements of a pair are separated by `key val sep`.

size (>"table", size>)= .

Affix `size` is given the number of pairs of `table`, in decimal notation.

lookup (>"table", >"key",value>)= "a".

`lookup` finds the first pair in the table with matching key. It fails if there is no such pair.

includes (>"table", >"key")= "a".
excludes (>"table", >"key")= "a".

Above predicates succeed if there is a (no) pair (K,P) in table L where L and K are the values of table and key respectively.

#plain
pair (cell>, >key,>value)= .
#plain
unpair (>"cell", key>,value>)= "a".

In addition to the table (tables) processing builtins there are three builtins which work on named tuples. Each tuple has a unique name. You can always work avoid using the predicates `pair` and `unpair`. Both are generated by eliminating the composition operator. For example the following two rules are equal:

```
b (>L1):
    unpair (>L1,a>,L2>),
    unpair (>L2,b>,c>).

b ( >a*b*c):.
```

If the left position is applying instead of defining `unpair` must be replaced with `pair`.

repair (>"cell", >key,>value)= "a";
repair (>"cell", >key,>value)= "a".

`repair` updates the earlier created cell (with `pair`) with new values. Upon backtracking this is undone unless deterministic behavior is forced. `repair` fails if there is no tuple `(key,value)` with name `cell`.

#plain
where (>x,x>)=.

Because the composition operator is right associative, brackets are needed to denote certain structures. Glammar does not allow you to write brackets in affix expressions. For example `>a*(x*y)*c` is not allowed. You can however make such structures using `where`. For example :

t (>a*b*c):
    my own where (>b,x*y>).

my own where (>x,x>):.

After the composition operator is removed :

t (>L1):
    unpair (>L1,a>,L2>),
    unpair (>L2,b>,c>),
    my own where (>b,L3>),
    unpair (>L3,x>,y>).

my own where (>x,x>):.

It is obvious that `my own where` has become superfluous if we substitute `b` for `L3`. This is the reason why there is a predefined built-in `where`. The compiler tries to remove `where` if it is possible by renaming affixes. So when used `where` instead of `my our own where` the result after transformation would have been :

t (>L1):
    unpair (>L1,a>,L2>),
    unpair (>L2,b>,c>),
    unpair (>b,x>,y>).

lexical less (>"x",>"y")= "a".
lexical more (>"x",>"y")= "a".
lexical less equal (>"x",>"y")= "a".
lexical more equal (>"x",>"y")= "a".

These builtins compare affixes using their lexical ordening.

compute (>expr, result>)= .

If `expr` is a string representing a simple expression (like `2*(3+1)`) `result` will assigned the value of this expression in decimal (`8`). The expression may only contain the one-character operators $! <>= + - */$ and signed integer notation. The maximum integer value is inherited from the C value for this. Booleans are implemented as $\{0, X\}$, where 0 denotes `false`, anything else (preferably 1) is taken to be `true`. Anything not an operator or integral number (spaces included) terminates the expression. A badly formed expression however gives an error message. For arbitrary precision arithmetic you can set up a connection to the unix program `dc`.

realtoint (>number, result>)= .

Converts a decimal real in scientific notation to the nearest decimal integer not greater than real.

decimal to unary (>number, result>)= .

Converts a decimal integer value to unary with base notation `1`.

identifier (x+y>)= !`AB..Za..z`! (x>), !`A..Za..z1..0'_`! * (y>).
letter digit sequence (x+y>)= !`AB..Za..z`! (x>), !`A..Za..z1..0`! * (y>).

These scan an identifier as specified but slightly more efficiently.

upper case (>str, result>)= .

`upper case` transforms a value with mixed cased letters to a upper cased value.

lower case (>str, result>)= .

`lower case` transforms a value with mixed cased letters to a lower cased value.

keyword (>"`upper case word`")= "`a`".

Matches an upper case word to lower or mixed case input. So keyword`(>"IF")` scans `if`, `IF`, `If`, and `iF`.

layout = !`\n\t\f` !*(ignored>).

You can use this rule to skip spaces, tabs, carriage returns, newlines, vertical tabs, and form feeds.

type out (>affix)=.

Write the value of affix to stderr.

type in (affix>)=.

Read affix from stdin.

fail = not equal (>empty,>empty).

`fail` just always fails.

#plain
cut= .

This predicate forces the next productions — if any — to fail. This member may also be written as `->`, and can be removed by adding complementary members to all other productions.

eval affix (>a,evaluated a>)= .

This predicate reduces the evaluation count at run time (see -r option) for deterministic grammars. Nondeterministic grammars overwrite the result of an affix evaluation and restore the original value upon backtracking. Since deterministic grammars do not need to backtrack the original value cannot be restored that easily. Instead, the result of the affix evaluation is lost which means that the same affix most be evaluated over and over again. If you wish to make your compiler as efficient as possible and you know that some affixes are tested frequently you can first evaluate these affixes and use the results in the rest of the computation.

set line file (>line,>file)= ;
set line file (>line,>file)= .
get line file (line>,file>)= ;
get line file (line>,file>)= .

Line and file control. In order to produce correct messages when the input was the output of a preprocessor there is a predicate which sets the current file name and line number.

get next option (option>)=.

This predicate gets the next option (the first one will be the command name) from the command line that invoked the program. It fails if all option are processed.

get env (>"name", val>)=.

`get env` searches the value of the environment variable. It fails if the environment variable is not defined.

error message (val>)=.

`error message` generates on error message based upon the most advanced input pointer. It causes to suppress output generation, and the exit status will be `1`. It can also be used to implement error recovery.

set exit code=.

This predicates sets the exit code to `2`, but output generation is not suppressed.

assign (>x,>y,old x>)=;
assign (>x,>y,old x>)=.

`assign` is a very tricky predicate and therefore its use is not recommended. Basically it makes it possible to avoid adding inherited affix positions if these affixes change in rare cases only. If this is the case you can let a metanotion take another value, although this sometimes give surprising results. For example in many cases it is necessary to copy the metanotion (when putting it in tables etc.), because of the fact that on an affix expression containing the metanotion, an assignation of that metanotion has effect on previous evaluations of that affix expression (consistent substitution is not guaranteed.)

get from unix (>cmd, cmd stdout>)=.

The most simple and reliable way to get some information from unix is to give a command, wait for the command to exit and store the result it produced. Examples of such unix utility programs are `date, cat, ls` etc. None of them need any input.

set up connection to unix (>cmd,>channel)=.
talk to unix (>channel,>to the programs stdin,from the programs stdout>)=.

These two builtins are only available on an unix-system. They can be used to set up and maintain a connection to unix. Only 32 connections can be made at a time. When this limit is exceeded `setup connection to unix` fails.

Useful programs to connect with are for instance `dc` for arbitrary precision arithmetic, or perhaps a program written by yourself. The lexicon interface also is implemented with connections.

As an example we will maintain a connection to `dc`. First we set up the connection.

s:
set up connection with unix (>"`dc `",>DC CON).

Here DC CON is a metadefined affix specifying the channel number. This channel is assigned to the metarule by the set up builtin; initially it may be defined as empty (eg : DC CON::.).

Next we can give input to this program's `stdin` and get the answer from the program's `stdout`. For example

x:
talk to unix (>DC CON , >"`1 2 +ps.\n`","3">).

will succeed. Because the builtin `talk to unix` first supplies input to the unix-program and then waits for the answer communication problems can occur. To solve this problem a very simple (not very reliable) strategy is used: As soon as a control character is read the answer reading is stopped and the result so far is made the total result. Caveat: the next time the builtin is called there might be some output left from previous calls. Also one has to take care that there always will be an answer. For instance in the above example the newline in the input given to `talk to unix` is essential because `dc` waits for it before generating any output, so both processes would have been locked forever ("deadlock") if this new line was omitted.

The following builtins are for internal usage and should therefore not be used in your own grammar. It is not possible to redefine these builtins and when its name is followed by an underscore it is also not possible to use the builtin yourself.

reset inputptr_ (>saved ip)= .
set inputptr to_ (>x,saved ip>)= .
nestarset (a>)= ;
nestarset (a>)= ;
nestarset (a>)= .
nestaralt_ (a>,>saved ip)= .
metaterminal (>"`m`")= "a".
intersect_ (>a,>b)= "a".
explintersect_ (>a,>b)= "a".
transform lattice_ (>a,>b,c>)= .
tl traditional_ (>a,>b,c>)= ;
tl traditional_ (>a,>b,c>)= ;
tl traditional_ (>a,>b,c>)= .
transform latticeterm_ (>a,>b,c>)= .
tl traditionalterm_ (>a,>b,c>)= ;
tl traditionalterm_ (>a,>b,c>)= ;
tl traditionalterm_ (>a,>b,c>)= .
init mint_ (>a,>b)= .

initmeta (>a,>b)= .
evalmeta (>a)= .
get ip_ (ip>)= "a".
restore ip_ (>ip)= "a".
false ip_ (>ip)= "a".
skip_= ;
skip_= ;
skip_= .

The following builtins implement multiple precision integer arithmetic. These are implemented using the unix standard package `mp`. Some implementations of unix do however not support this package. More information can be found in the manual page for `mp`.

equal to ::"0".
less than ::"-1".
more than ::"1".

int (>a) **plus int** (>b) **is int** (c>)=.
int (>a) **sub int** (>b) **is int** (c>)=.
int (>a) **times int** (>b) **is int** (c>)=.
int (>a) **divided by int** (>b) **is int** (q>) **with remainder int** (r>)=.
int (>a) **is** (c>) **int** (>b)=.
int (>a) **raised by int** (>b) **modulo int** (>d) **is int** (r>)=.
int (c>) **is the gcd of int** (>a) **and int** (>b)=.
int (s>) **is the square root of int** (>a) **with remainder int** (r>)=.
int (>a) **is dec** (dec a>)=.
dec (>a) **is int** (int a>)=.
free int (>d)=.
decrement small int (>d)=.
increment small int (>d)=.
small dec (>d) **is int** (i>)=.
int (>a) **is negative**=.
negate int (>a) =.
int (>a) **is positive**=.
int (>a) **is zero**=.

dec is exact (>Dec Is Exact, exact pq>)=.
normalize exact (>p, >q, np nq>)=.
exact is dec (>p q, dec exact>)=.
exact (>ap aq) **plus exact** (>bp bq) **is exact** (result>)=.
exact (>ap aq) **sub exact** (>bp bq) **is exact** (result>)=.
exact (>ap aq) **times exact** (>bp bq) **is exact** (result>)=.

exact (>ap aq) **divided by exact** (>bp bq) **is exact** (result>)=.
exact (>a) **is** (c>) **exact** (>b) =.
exact number (exact>)=.

10. Manual page

NAME

GLAMMAR — EAG based compiler compiler
Version 1.00 Oktober 1990

AUTHOR

Eric Voss , KU Nijmegen A3017, tel: (080-65)2306, ericv@cs.kun.nl

SYNOPSIS

GLAMMAR [options] [file—-]

DESCRIPTION

GLAMMAR is an *EAG* (Extended Affix Grammar) compiler compiler entirely written in C. GLAMMAR might be something for problems you use to solve with *PROLOG, YACC* etc.

The source program is specified as an extension of context free grammars. This extension involves affixes. Results are reported as in YACC by derived affixes. Context sensitive checks can be implemented using inherited affixes.

GLAMMAR generates a program (compiler) `prog` from a source grammar `prog.g` or reads a grammar from `stdin` and writes C-code for the compiler to `stdout` (-) depending on the arguments. Also an intermediate file `file_.c` is created and removed upon completion of the generation process.

If a compiler is generated (lets assume you typed `glammar prog`) you can give the input and output files and options to the generated compiler in the same way (i.e typing: `MyGrammar in out` or just `MyGrammar` will read from the file `in` or `stdin` and write output to the file `out` or `stdout` respectively).

GLAMMAR OPTIONS

The following options are interpreted by GLAMMAR :

-a: Let the generated compiler dump a message for *ambiguous* each ambiguous rule.
-d: Force some (but not all) builtins to behave *deterministically.* If the start rule is not deterministic a listing is given of nondeterministic rules and compilation is aborted.
-mn: *memorizing.* This option specifies to memoize productions that do not use inherited affixes (if any) in the remaining nondeterministic part (thus for instance after the builtins (equal, not equal)) provided this remaining tail does not produce empty. Currently grammars compiled with this option have a limit in respect to the input size (nK symbols). The default size of n is 2.
-Mn: Advanced memorizing. This option specifies that the grammar

will be transformed to a weak equivalent form in which productions are either *empty only* or *not empty.*

-t: Simple *tracer.* The execution of the generated compiler will be traced. This trace consists of a dynamic level number, the current hyperrule, and the inherited affixes of the current hyperrule. Undefined affixes are printed as `!<@>!`, composed affixes are unpaired once, remaining compositions are printed as `!<adress>!` where `address` is some decimal number.

-T: Same as -t but builtins are also traced.

-p: Take the input file name to be a *part-list* (a '.p' file) of subsequent input files (see `separate compilation`). Compiles all those parts that are out of date. If the local directory `.glammar` exists, object files are placed in this directory.

-P: Same as -p.

-oname: Pass name as an option to `cc` or `gcc`. The `o` is stripped. For example: `glammar -ofomit-frame-pointer -oO eag`.

-x: Produce an *index* file ('.x') of defined hyper- and metarules with file name and line number. The index file is not sorted.

-X: Make an index-file only (no compilation).

-f: Do not *lift* certain (nondeterministic) builtins which are in front of a production rule.

-g: Do not use the *Gnu* `gcc` compiler.

-s: Produce a `.c` file only.

-V: Be *verbose*

-v: Print the *version* number and installation date.

-r[trans:] If `trans` is omitted then after all the transformations are performed, compilation is aborted and the *transformed* grammar is *printed* to stdout. Transformations are done in the order: Composition operator elimination, defining expression elimination, implied equal elimination. If `trans` is `idx` and the grammar is a context free grammar this option generates a new grammar with affixes added such that the new grammar will output what it recognized. This saves some work if you write a transducer by the following steps: first write a context free grammar, compile this grammar using option `-ridx` and adapt the displays in this output to conform your needs.
If `trans` is `parse-tree` and the grammar is a context free grammar this option generates a new grammar with affixes such that when this grammar is recompiled, a parse-tree is generated for the original grammar. If `trans` is `cfg` the grammar is deprived of its displays.

-cDirname: *Hackflag (hackers only).* This flag is most useful for those who want to *extend the set of builtins.* `Dirname` is the full path name of the directory where your own file `glext.g` should exist. Furthermore in this same directory a file `glext.o` is assumed which is linked together with the system files to form an executable compiler. For information on how to organize your own builtins see the description of the C interface.

GENERATED COMPILER OPTIONS

The following options are interpreted by the generated program.

-an: Limit the number of parses to n.
-E: Do not give *error messages* but exit with 1.
-r: Message resource usages like stack size and affix store size.
-hn: Increase affix store size (used for deterministic grammars) (n = 1 means 2 megabytes).
-cn: Increase character store size (n=1 means 512k).
-qn: Increase continuation stack size (n=1 means 512k).
-l: Line loop mode. That is read and process one line at a time. Each line is taken to be a single input sentence.
-tn: If compiled with the trace option , begin tracing at dynamic level n. If n equals -1 tracing is completely suppressed.
–: Do not read the input.

DIAGNOSTICS

The diagnostics produced by GLAMMAR are self explaining.

SPECIAL FEATURES

Read the manual for a full description of GLAMMAR If you are familiar with EAG, some special features of GLAMMAR are:

cut operator on hyper level: The cut operator is implemented as a special builtin, namely `cut`. This may be abbreviated with `->` though by doing so syntax looks awful (e.g. `a,->;` stands for `a,cut;`) When this operator is applied, next alternatives are omitted. This operator does not fundamentally affect the semantics of the GLAMMAR system, since each grammar can still be transformed to an equivalent one without this operator.

underscore (_) as don't care affix name : Each underscore is replaced at compile time to something like `x_%X` where `%X` is a unique hexadecimal number during parsing. Therefore underscores on applying positions result in an error message of the form : affix `x\_1` not defined in ... .

super rules: Super rules are a special kind of metarules. At this moment they only are allowed on defining positions.
Superrules are declared as : `super :: @ hyper.`
`hyper` should have a derived affix (if any). So with superrules you can mix metalevel and hyperlevel and affix definitions can indirectly have alternatives.

connection to unix programs.: Several builtins have been written to maintain a connection during the execution of your compiler to some other program. These builtins are `set up connection to unix` and `talk to unix`. See the manual.

additional builtins: The set of builtins is extended with builtins that trace affix values, do integer arithmetic, line and file control in error messages, fast symbol scanning etc. For a complete listing you can print out the file `$LIB/glammar.g` (You may prefer to format it first with the pretty printer `glammar-pp`).

Finite Lattices : Lattices can be used as a very efficient unification tool. See the manual.

SEE ALSO

glammar-pp, cc

For a practical introduction: Eric Voss, GLAMMAR *user's and reference manual* Internal paper, 1990

BUGS

- Metarules can only have one production. Metarules that need multiple productions can perhaps be simulated using *super rules* or else *lattices*
- No delayed affix evaluation.
- No left recursion on hyperlevel.

Send unmentioned bugs or just comments to ericv@cs.kun.nl.

11. Pretty printer

NAME

glammar-pp – prototype GLAMMAR source pretty printer.
Version 1.0. November 1990.

AUTHOR

Eric Voss , KU Nijmegen ericv@cs.kun.nl

SYNOPSIS

glammar-pp infile outfile

DESCRIPTION

Glammar-pp is a prototype of a GLAMMAR source pretty printer written in GLAMMAR and producing a LaTeXsource file.

The pretty printer takes comment starting with `TEX`, `^TEX` and `$TEX` to be LaTeX command input. Note that GLAMMAR will ignore all text between `%`, so you can still compile the same program without the need to make a correct source file first. The first directive has to be `^TEX`, the last `$TEX`, others `TEX`.

At the beginning of the GLAMMAR source file you should put some LaTeX stuff like the specification of a document style that will include the GLAMMAR style file `glammar.sty` and like author etc. The following example provides a suitable header:

```
%^TEX
\documentstyle[11pt,glammar]{article}
\author{Eric Voss  KUN}
\title{Pretty printer for EAG\\ Version 1.0}
\parskip 0pt plus 1pt
\parindent 0pt
\begin{document}
%
```

While a suitable tailer would be:

```
%$TEX
\end{document}
%
```

Experienced LaTeX users will have no trouble in producing a well formatted document but if you never have used LaTeX here are some hints:

- Separating your grammar in sections can be done by typing:

  ```
  %TEX
  \section{Syntax.}
  %
  ```

- Index: An index can be included by running `latex`, `makeindex` and `latex` again after including the following text at the appropriate place text in your document:

  ```
  #input \jobname.ind
  ```

"SEE ALSO"

glammar(1), LaTeX(1).

CHAPTER IV

Glammar Interface

1. Interface to unix

The builtins `get from unix, talk to unix, setup connection to unix` can be used to get some information from unix or maintain a connection to a program by writing to the program's input and reading the result from the program's output. Further information can be found in the description of these builtins.

2. C-language interface

The runtime system of GLAMMAR consists of routines implementing the standard definitions. Those who want to extend this set should examine some members of it carefully.

For each deterministic builtin `builtin` two routines must be written, a deterministic `dbuiltin(affix1, affix2 ..  )` and a nondeterministic one with no arguments `ubuiltin()`. For nondeterministic builtins only the nondeterministic one needs to be given.

As an example we show how to make a new deterministic builtin `mybuiltin`. Let us assume this builtin takes two affixes, an inherited and a derived one. First we make an empty rule: `builtin(>expr, empty>):.` This builtin must (upon usage) be stored in `glext.g`. In the file `glext.g` some simple nondeterministic grammar must be given in order to let GLAMMAR generate both variants. For example the file `glext.g` may have the following contents:

```
# glext.g
dummy (empty>):;.

builtin(>a,"a">):.
```

Compiling (i.e. `glammar -s -m`) the grammar (with the memorize option!) gives a frame in which we change and add some lines.

```
/*  builtin   */
int dbuiltin(A_0,A_1)
struct affix *A_0;
```

```
struct affix *A_1;
{
   register struct affix *raf = af;
   char *rc = c,  *rip = ip;
   {
         A_1 -> t = "a";
         A_1 -> l = nil;
         A_1 -> r = nil;
         return true;
   }
   return false;
}

/* non deterministic part */
void ubuiltin () {
   register cont *rq = q;
   register struct affix *A_1 = (rq+0)-> a;
   register struct affix *A_0 = (rq+-1)-> a;
   if (dbuiltin(A_0,A_1))
   {
      q=rq+-3;
      (*(rq+-2)->q)();
      rq=q- -2;
   }
   (rq+-1)->a = A_0;
   (rq+0)->a = A_1;
   (rq+1)-> q = ubuiltin;
   q = rq+1;
}
```

Let us assume a C routine `int builtin(char *in, char *out)` is written which expresses the semantics of `builtin`. We change the above to:

```
/*  builtin   */
int dbuiltin(A_0,A_1)
struct affix *A_0;
struct affix *A_1;
{
   register struct affix *raf = af;
   char *rc = c,  *rip = ip;
   {
+        char *in,*out;

+        if (A_0 -> l == nil && A_0 -> r == nil)
+           in = A_0 -> t;
+        else {
+           in =  c;
+           sprinta (A_0);
+           *c++ ='\0';
+        }
+        if (builtin(in,out)) {
```

```
+           A_1 -> t = out;
            A_1 -> l = nil;
            A_1 -> r = nil;
            return true;
+        }
   }
   return false;
}
```

The first part we inserted evaluates the inherited affix. This is only done when necessary (i.e. neither the left and the right part is nil). In general inherited affixes are evaluated first via a call to `sprinta(afx)`. The result is stored on the affix heap which is just a row of characters. It contains only results of affixes. We have named this heap the `cstore`; the cstore-pointer `c`. For example if we want to evaluate an affix and name the result `arg`, we have to execute the following program fragment:

```
char  *arg = c;
sprinta(afx);
*++c = '\0';
```

If the function `sprinta(afx)` is called, `afx` has to be defined , otherwise a runtime error will occur.

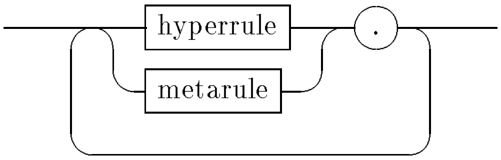
Next we can let `builtin` assign the result to `A_1` if the builtin succeeded. Suppose a result (the string `result`) is obtained which has to be assigned to the derived affix A1:

```
A1 -> t = result; A1 -> l = nil ; A1 -> r = nil;
```

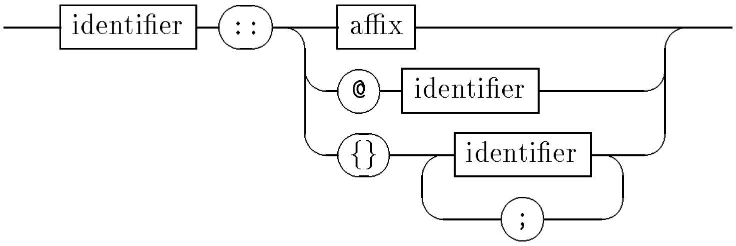
Appendices

Appendix A. The syntax of Glammar

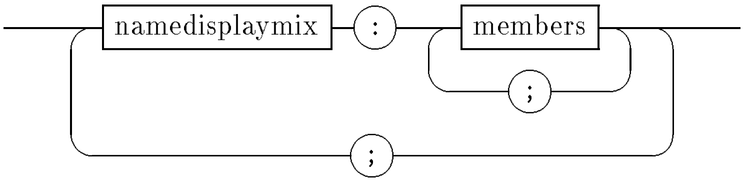
glammar



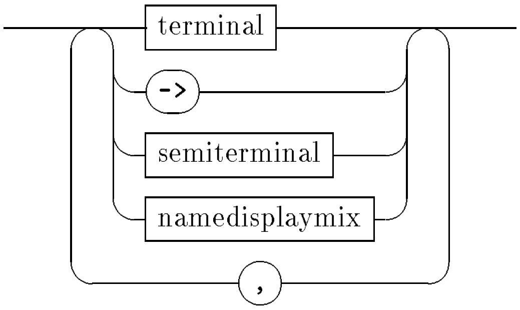
metarule



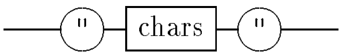
hyperrule



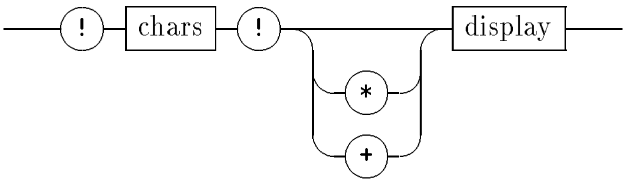
members



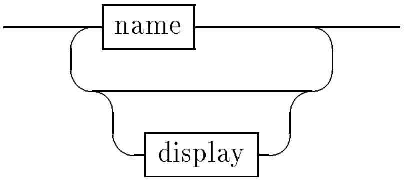
terminal



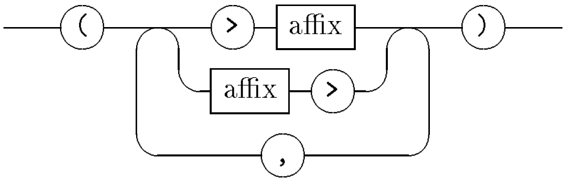
semiterminal



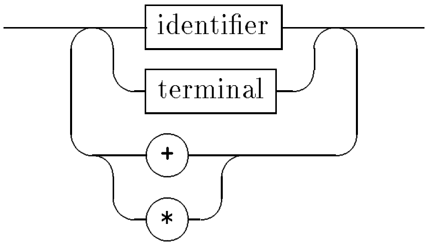
namedisplaymix



display



affix



Appendix B. Examples

B.1. Eight Queens. In the following example we will use the composition to make a list structure. This example — as it stands — solves the eight queens problem. It is not a representive example in GLAMMAR because it only contains predicates (it reads no input).

```
eight queens on a chess board ("Look what I found :  "+nlcr+y+nlcr>):  layout,
    permute (>Eight Queens,x>),
    safe (>x),
    walk (>x,y>).

permute (>a*list,out>) :
    permute (>list,another list>),
    gen (>a, >another list, out>).
permute (>empty,empty>) : .

# 'gen' is the only direct non-deterministic rule.
gen (>a,>list, a*list>):.
gen (>a, >b*list, b*out>):
    gen (>a, >list, out>).

safe (>a*b):  safe (>b),no diag (>a,>One,>b).
safe (>empty):.

no diag (>a ,> m, >c*d):
    where (>a , a minus c + c>),
    not equal (>a minus c ,>m),
    no diag (>a,>m+One,>d).
no diag (>a ,> m, >c*d):
    where (>c, c minus a + a>),
    not equal (>c minus a ,>m),
    no diag (>a,>m+One,>d).
no diag (>a,>n, >empty): .

walk (>a*b,out+nlcr+wb>):
    out (>a, out>),
    walk (>b, wb>).
walk (>empty,empty>):.

out (>One,  " q  .  .  .  .  .  .  .">):.
out (>Two,  " .  q  .  .  .  .  .  .">):.
out (>Three," .  .  q  .  .  .  .  .">):.
out (>Four, " .  .  .  q  .  .  .  .">):.
out (>Five, " .  .  .  .  q  .  .  .">):.
out (>Six,  " .  .  .  .  .  q  .  .">):.
out (>Seven," .  .  .  .  .  .  q  .">):.
out (>Eight," .  .  .  .  .  .  .  q">):.

Eight Queens :: One*Two*Three*Four*Five*Six*Seven*Eight*empty.
```

```
One   :: "i".
Two   :: "i" +One.
Three :: "i" +Two.
Four  :: "i" +Three.
Five  :: "i" +Four.
Six   :: "i" +Five.
Seven :: "i" +Six.
Eight :: "i" +Seven.
```

B.2. Towers of Hanoi. The following example solves the famous problem of moving discs with different sizes from one tower to another using a third without placing a large disc on a smaller one.

```
hanoi ( moves>) :
    number (N>),
   (moves>) moves (>N) disks from (>"S") to (>"D") using (>"A").

("Move disc 1 from " +S + " to " +D + nlcr>)
moves (>one or less) disks from (>S) to (>D) using (>no A):
    int (>one or less) is (less than>) int (>TWO) ,—>;

(x+"Move disc "+dec N+ " from " + S+ " to "+ D+ nlcr+y>)
moves (>N) disks from (>S) to (>D) using (>A):
    int (>N) is  dec (dec N>),
    int (>N) sub int (>ONE) is int (M>),
   (x>) moves (>M) disks from (>S) to (>A) using (>D),
   (y>) moves (>M) disks from (>A) to (>D) using (>S).

number (int n>):
    layout, !0123456789! + (string n>),layout,
   (>string n) dec is int (int n>).

TWO::2.
ONE::1.
```

B.3. Pizza. This grammar transforms a sequence of letters, representing the ingredients of a pizza, to the name of that pizza containing at least the ingredients. For example, input `ctb` produces the output:

```
take a Borromea
take a QuattroStagioni
take a Ciociara
```

The first version of this grammar is:

```
what is ("take a "+pizza+nlcr>):
   pizza (pizza>),
   more pizza ( pizza> ),layout.

more pizza (pizza>):
```

  pizza (pizza>),
  more pizza (pizza>);
more pizza (pizza>):
  all (pizza>).

pizza (C>): "c",cheese(C>);
pizza (O>): "n",onion(O>);
pizza (O>): "l",olives(O>);
pizza (A>): "a",anchovy(A>);
pizza (T>): "t",tomato(T>);
pizza (B>): "b",bacon(B>);
pizza (E>): "e",eggs(E>);
pizza (M>): "m",mushroom(M>);
pizza (X>): "*",all(X>).

cheese (Borromea>):; cheese(Siciliana>):; cheese(QuattroStagioni>):; cheese(Ciociara>):.

onion (Borromea>):; onion(QuattroStagioni>):; onion(Ciociara>):.

olives (Siciliana>):.

anchovy (Siciliana>):.

tomato (Borromea>):; tomato(Siciliana>):; tomato(QuattroStagioni>):; tomato(Ciociara>):.

bacon (Borromea>):; bacon(QuattroStagioni>):; bacon(Ciociara>):.

eggs (Ciociara>):.

mushroom (QuattroStagioni>):; mushroom(Ciociara>):.

all (Borromea>):; all(Siciliana>):; all(QuattroStagioni>):; all(Ciociara>):.

Borromea :: "Borromea".
Siciliana :: "Siciliana".
Ciociara :: "Ciociara".
QuattroStagioni :: "QuattroStagioni".

The problem of this grammar is its exponential time behavior. On input

`a*************************`

we must wait almost forever for the answer. It takes a top-down backtracking program 4^30 possibilities to analyze before the result is found. Even memorization does not help because it only cuts paths that failed earlier. Here the search space is just too big to try the possible solutions one by one.

This problem also appears in practice for linguistic grammars. Thinking about this problem one recognizes soon that the problem occurs because we initially have a set of possible values and — while parsing — context conditions do not allow all possible combinations. The computation starts with one possible value and (when it fails) another solution is tried beginning all over again. The solution is obviously to begin the computation with the whole set and when a consistent substitution is required to go on with the intersection of the two sets. In this way the computation only fails if there are no elements left in the set. In GLAMMAR this idea is implemented by representing the power set (i.e. the set of all intersections over a type) by a finite bit sequence. Hence intersection can be implemented very efficiently. The number of elements however is limited.

Lattices are not automaticly recognized by GLAMMAR . Quite the opposite holds actually. In the grammar, both application and definitions of affixes that are represented by lattices must omit the flow symbol and use a special metanotation respectively. We will explain this by rewriting the previous pizza example.

```
what is ("take a "+PIZZA+nlcr>):
   pizza (PIZZA),
   more pizza ( PIZZA ).

more pizza (PIZZA):
  pizza (PIZZA),
  more pizza (PIZZA);
more pizza (PIZZA):.

pizza (CHEESE):     "c";
pizza (ONION):      "n";
pizza (OLIVES):     "l";
pizza (ANCHOVY):    "a";
pizza (TOMATO):     "t";
pizza (BACON):      "b";
pizza (EGGS):       "e";
pizza (MUSHROOMS):  "m";
pizza (PIZZA):      "*".

PIZZA :: {} CHEESE; ONION; OLIVES; ANCHOVY; TOMATO;
            BACON; EGGS; MUSHROOMS.

CHEESE :: {} Borromea; Siciliana; Quattro Stagioni; Ciociara.

ONION :: {} Borromea ; Quattro Stagioni; Ciociara.

OLIVES :: {} Siciliana .

ANCHOVY :: {} Siciliana .

TOMATO :: {} Borromea; Siciliana; Quattro Stagioni; Ciociara.

BACON :: {} Borromea; Quattro Stagioni; Ciociara.

EGGS :: {} Ciociara.

MUSHROOMS :: {} Quattro Stagioni; Ciociara.
```

This version proccesses the input `a**************************` almost instantaneously.

B.4. Implementation of exact numbers. This is the source grammar which is used to implement the builtins operating on exact numbers.

```
#Exact numbers
dec is exact (>Dec Is Exact, exact pq>)=
     where (>Dec Is Exact,pp*qq>),
     normalize exact (>pp, >qq, exact pq>),
     free int (>pp),
     free int (>qq).

normalize exact (>p, >q, exact>)=
     int (>p) is negative,
     int (>q) is positive,
     negate int (>p),
     more normalize exact (>p, >q, exact>),
     where (>exact, np*nq>),
     negate int (>np), —>;
normalize exact (>p, >q, exact>)=
     more normalize exact (>p, >q, exact>).

more normalize exact (>p, >q, np*nq>)=
     int (g>) is the gcd of int (>p) and int (>q),
     int (>p) divided by int (>g) is int (np>) with remainder int (rp>),
     int (>q) divided by int (>g) is int (nq>) with remainder int (rq>),
     free int (>rp),
     free int (>rq),
     free int (>g).

exact is dec (>p*q, dec exact>)=
    int is dec (>p,dec p>),
    int is dec (>q,dec q>),
    beautify dec exact (>dec p, >dec q, dec exact>).

beautify dec exact (>p, >"1", p>): —>;
beautify dec exact (>p, >q, p + "/" + q>):.

Dec Is Exact ::@ exact number.

exact (>ap*aq) plus exact (>bp*bq) is exact (result>)=
```

```
    int (>ap) times int (>bq)  is int (apbq>),
    int (>bp) times int (>aq)  is int (bpaq>),
    int (>aq) times int (>bq)  is int (q>),
    int (>apbq) plus int (>bpaq) is int (p>),
    normalize exact (>p,>q,result>),
    free int (>p),
    free int (>q),
    free int (>apbq),
    free int (>bpaq).

exact (>ap*aq) sub exact (>bp*bq) is exact (result>)=
    int (>ap) times int (>bq)  is int (apbq>),
    int (>bp) times int (>aq)  is int (bpaq>),
    int (>aq) times int (>bq)  is int (q>),
    int (>apbq) sub int (>bpaq) is int (p>),
    normalize exact (>p,>q,result>),
    free int (>p),
    free int (>q),
    free int (>apbq),
    free int (>bpaq).

exact (>ap*aq) times exact (>bp*bq) is exact (result>)=
    int (>ap) times int (>bp)  is int (p>),
    int (>aq) times int (>bq)  is int (q>),
    normalize exact (>p,>q,result>),
    free int (>p),
    free int (>q).

exact (>ap*aq) divided by exact (>bp*bq) is exact (result>)=
    int (>ap) times int (>bq)  is int (p>),
    int (>aq) times int (>bp)  is int (q>),
    normalize exact (>p,>q,result>),
    free int (>p),
    free int (>q).

exact (>a) is (c>)  exact (>b) =
   exact (>a) sub exact (>b) is exact (p*q>),
   free int (>q),  # q is made zero by  free int
   int (>p) is (c>)  int (>q) ,
   free int (>p).


exact number (exact>)=
   !-0123456789! +(d>),
   rest exact number (>d,exact>).

rest exact number (>dp,p*i>):
   "/", dec (>dp) is int (p>),
   !0123456789! +(dq>), not equal (>dq, >"0"),dec is int (>dq,i>), —>;
```

```
rest exact number (>p,e>):
    ".",
    !0123456789! +(d>),
    compute exact (>p,>d,e>),—>;
rest exact number (>dp,p*q>):
    dec is int (>dp,p>),
    dec is int (>"1",q>).

compute exact (>dp,>d,e>):
    how much zeros (>d,zz>),
    dec is int (>"1"+zz,q>),
    dec is int (>dp+d,p>),
    normalize exact (>p, >q , e>).

how much zeros (>C+r,"0"+z>):
    how much zeros (>r,z>), —>;
how much zeros (>empty,empty>):.

C :: @ one char.

one char (i>): ^!!(i>).
```

B.5. Palindromes. The following grammar recognizes palindromes. A palindrome is a sentence from the language $L(p) = \{w + v + w^R||w| > 0, |v| \leq 1, |w| + |v| > 0, w, v \in A\}$ where A is the set of symbols that are allowed in terminals. In the following grammar A is the set of all characters that are allowed in GLAMMAR .

```
pal : one char (c>), pal, one char (c>),—>; one char (x>);.

one char (c>): ^!!(c>).
```

B.6. Sequences. The following examples show how all kinds of sequences can be described in a convenient and efficient manner.

```
#A possibly empty sequence with no separator symbol we can efficiently describe by:
l*: l, l*;
l* :.

#Such a sequence with at least one element:

l+: l, l*.


#Such a sequence with at least two elements:

l++: l, l+.
```

#A possibly empty sequence with a separator symbol we can efficiently describe by :

k*: k, k'*;
k*:.

k'*: k seq separator, k, k'*;
k'*:.

#Such a sequence with at least one element :

k+: k, k'+.

k'+: k seq separator, k+;
k'+:.

#Such a sequence with at least two elements :

k++: k, k seq separator, k, k'+.

B.7. An interactive calculator. This advanced example shows that programs can be interactive, and how to setup a connection to talk to a program. Finally it shows how infix operators with different priorities can be implemented efficiently.

simple calculator:
    prologue,
    statement sequence,
    type out (>"\nType ^C to quit\n?"),
    loop.

statement sequence:
    end of sentence, —>;
statement sequence:
    statement, type out (>nlcr),
    statement sequence.

loop:
    type in (STATEMENT>),
    loop.

STATEMENT :: @ metastatement.
metastatement (empty>): statement, type out (>"\n?").

statement:
    "PRINT", layout, expr (exp>),
    dc (>exp +" ps.", r>),
    type out (>r), —>;

```
statement:
        "REM", ^!\n!* , nlcr, —>;
statement:
        "LET", layout, !abcdefghijklmnopqrstuvwxyz! (id>), layout,
        "=", layout, expr (val>),
        dc (>val+" s"+id +" p",_>), —>;
statement:
        ^!! *, type out (>"Syntax error").

dc (>x,y>):
        talk to unix (>DC, >x+nlcr,y>).


expr (exp>):
        e1 (>empty, >empty, >empty, exp>).

e1 (>old s, >old p, >old prio, "0 "+e1>):
        prefix (prefix>, key>, new prio>),
        associate (>empty*old s, >old p, new s>, new p>, >old prio, >key, >prefix),
        e1 (>new s, >new p, >new prio, e1>), —>;
e1 (>old s, >old p, >old prio, exp>):
        simple (simple>),
        e1 rest (>old s, >old p, >old prio, >simple, exp>).

e1 rest (>old s, >old p, >old prio, >oper, exp>):
        infix (infix>, key>, new prio>),
        associate (>oper*old s, >old p, new s>, new p>,>old prio,>key,>infix),
        e1 (>new s, >new p, >new prio, exp>),—>;
e1 rest (>old s, >old p, >old prio, >oper, exp>):
        associate (>oper*old s, >old p, exp*_>, new p>, >old prio, >empty, >empty).

associate (>right s*left s* ss, >oper*prio*p,
                        new s>, new p>, >old prio, >key, >infix):
        compare (>old prio, >key, rest o>, rest k>),
        not equal (>empty, >rest o), —>,
        where (>" "+left s +" "+right s+ " "+ oper ,s>),
        associate (>s*ss, >p, new s>, new p>, >prio, >key, >infix);
associate (>old s, >old p, old s>, infix*old prio *old p>,
                        >old prio, >key, >infix):.


simple (n>):
        number (n>), —>;
simple (" l"+l+" ">):
        !abcdefghijklmnopqrstuvwxyz! (l>), layout,—>;
simple (x>):
        "(", layout, expr (x>), ")", layout.

number (x>):
```

```
    !01223456789! + (x>), layout.

infix (i>,prio>,key>):
    !+-%^*/! (x>),
    lookup (>x,>INFIX,i*prio*key>), layout.

prefix (p>,prio>,prio key>):
    !-+! (x>),
    lookup (>x,>PREFIX,p*prio*prio key>), layout.

INFIX :: "i".
PREFIX :: "p".

prologue :
    setup connection to unix (>"dc", >DC),
    define infix (>"+",     >one,>one),
    define infix (>"-",     >one,>one),
    define infix (>"*",     >two,>two),
    define infix (>"/",     >two,>two),
    define infix (>"%",     >three,>three),
    define infix (>"^",     >five,>six),
    define prefix (>"-",    >four,>four),
    define prefix (>"+",    >four,>four).

define infix(>infix,>prio left,>prio right):
    add to (>infix,>INFIX,>infix*prio left*prio right).

define prefix(>prefix,>prio left,>prio right):
    add to (>prefix,>PREFIX,>prefix*prio left*prio right).

zero::      "".
one::       "1".
two::       "11".
three::     "111".
four::      "1111".
five::      "11111".
six::       "111111".

DC::.
```

Appendix C. Builtins

The following figure contains all builtins, numbers refer to the page numbers of relevant descriptions in section III. 9

addto 60	assign 65	backslash 58
charnum 59	compare 59	compute 63
contentsof 61	cut 64	decimaltounary 63
decisexact 67	decisint 67	decrementsmallint 67
delete 60	empty 58	endofsentence 59
equal 59	equalto 67	errormessage 65
evalaffix 64	evalmeta 66	exactdividedbyexactisexact 67
exactisdec 67	exactisexact 67	exactnumber 67
exactplusexactisexact 67	exactsubexactisexact 67	exacttimesexactisexact 67
excludes 61	explintersect 66	fail 64
falseip 66	freeint 67	generationdate 58
generationtime 58	getenv 65	getfromunix 65
getip 66	getlinefile 64	getnextoption 64
identifier 63	includes 61	incrementsmallint 67
initmeta 66	initmint 66	intdividedbyintisintwithremainderint 67
intersect 66	intisdec 67	intisint 67
intisnegative 67	intispositive 67	intisthegcdofintandint 67
intisthesquarerootofintwithremainderint 67	intiszero 67	intplusintisint 67
intraisedbyintmodulointisint 67	intsubintisint 67	inttimesintisint 67
keyword 63	layout 63	length 59
lessthan 67	letterdigitsequence 63	lexicalless 62
lexicallessequal 62	lexicalmore 62	lexicalmoreequal 62
linenum 59	lookup 61	lowercase 63
metaterminal 66	morethan 67	negateint 67
nestaralt 66	nestarset 66	nlcr 58
nlcr 58	normalizeexact 67	notequal 59
pair 61	quote 58	quote 58
realtoint 63	repair 61	resetinputptr 66
restoreip 66	select 60	setexitcode 65
setinputptrto 66	setlinefile 64	settabstop 59
setupconnectiontounix 66	size 61	skip 66
smalldecisint 67	somename 60	talktounix 66
tltraditional 66	tltraditionalterm 66	transformlattice 66
transformlatticeterm 66	typein 64	typeout 63
unpair 61	uppercase 63	valueof 60
where 62		

FIGURE IV.3. List of builting

Distribution and License

Distribution

GLAMMAR is free; this means that everyone is free to use it and free to redistribute it on a free basis. GLAMMAR is not in the public domain; it is copyrighted and there are restrictions on its distribution—restrictions similar to those of GNU software.

The easiest way to get a copy of GLAMMAR is to ftp the sources from `phoibos.cs.kun.nl [131.174.81.1]`. You do not need permission. If you cannot get a copy this way, you can order one from University Computing Services. Though GLAMMAR itself is free, our distribution service is not. For further information, contact

> Prof.dr.ir. R. Boute
> Computer and Communication Systems group
> Department of Computer Science,
> University of Nijmegen
> Toernooiveld 1
> 6525 ED Nijmegen
> The Netherlands

Glammar General Public License

Out intention is to give everyone the right to share GLAMMAR . To make sure that you get the rights we want you to have, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. Hence this license agreement.

For our own protection, we must make certain that everyone finds out that there is no warranty for GLAMMAR .

Copying Policies

(1) You may copy and distribute verbatim copies of GLAMMAR source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each file a valid copyright notice "Copyright © 1991 University of Nijmegen" (or with whatever year is appropriate); keep intact the notices on all files that refer to this License Agreement and to the absence of any warranty; and give any other recipients of GLAMMAR a

copy of this License Agreement along with the program. You may charge a distribution fee for the physical act of transferring a copy.

(2) You may modify your copy or copies of GLAMMAR source code or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:
  - cause the whole of any work that you distribute or publish, that in whole or in part contains or is a derivative of GLAMMAR or any part thereof, to be licensed at no charge to all third parties on terms identical to those contained in this License Agreement (except that you may choose to grant more extensive warranty protection to some or all third parties, at your option).
  - You may charge a distribution fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

  Mere aggregation of another unrelated program with this program (or its derivative) on a volume of a storage or distribution medium does not bring the other program under the scope of these terms.

(3) You may copy and distribute GLAMMAR (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:
  - accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,
  - accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal shipping charge) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,
  - accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

  For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs.

(4) You may not copy, sublicense, distribute or transfer GLAMMAR except as expressly provided under this License Agreement. Any attempt otherwise to copy, sublicense, distribute or transfer GLAMMAR is void and your rights to use GLAMMAR under this License agreement shall be automatically terminated. However, parties who have received computer software programs from you with this License Agreement will not have their licenses terminated so long as such parties remain in full compliance.

Bibliography

[1] H. Alblas, *Pass-oriented attribute evaluation and attributed tree transformations*, Ph.D. thesis, Universiteit Twente, Enschede, The Netherlands, 1987.

[2] Henk Alblas, *Optimal incremental simple multi-pass attribute evaluation*, Memorandum INF-86-27, Twente University of Technology, August 1986, Also in [1].

[3] Isabelle Attali and Paul Franchi-Zannettaci, *Unification-free execution of TYPOL programs by semantic attribute evaluation*, Rapports de Recherche 864, INRIA, July 1988.

[4] B. Courcelle, *Attribute grammars: Definitions, analysis of dependencies, proof methods*, Methods and Tools For Compiler Construction (B. Lorho, ed.), Cambridge University Press, pp. 81–102.

[5] Bruno Courcelle and Paul Franchi-Zannettacci, *Attribute grammars and recursive program schemes II*, 235–257.

[6] D. Crowe, *Generating parsers for Affix Grammars*, no. 8, 728–734.

[7] Alan J. Demers, *Generalized left corner parsing*, Fourth ACM Symposium on Principles of Programming Languages, pp. 170–182.

[8] Pierre Deransart, Martin Jourdan, and Bernard Lorho, *Attribute grammars : definitions, systems, and bibliography*, Lecture Notes in Computer Science, vol. 323, Springer-Verlag, Berlin, 1988.

[9] J. Earley, *An efficient context-free parsing algorithm*, Ph.D. thesis, University of Pittsburgh, 1968.

[10] P. Franchi-Zannettacci, *Attribute specifications for graphical interface generation*, Rapports de Recherche 937, INRIA, December 1988.

[11] H. Ganzinger and R. Giegerich, *Attribute coupled grammars*, ACM SIGPLAN Symp. on Compiler Construction.

[12] Robert Giegerich, *Composition and evaluation of attribute coupled grammars*, 355–423.

[13] Susan L. Graham, Michael A. Harrison, and Walter L. Ruzzo, *An improved context-free recognizer*, no. 3, 415–462.

[14] Mehdi Jazayeri and Diane Pozefsky, *Space-efficient storage management in an attribute grammar evaluator*, no. 4, 388–404.

[15] T. Katayama, *Translation of attribute grammars into procedures*, no. 3, 345–369.

[16] Donald E. Knuth, *Semantics of context-free languages*, Mathematical Systems Theory **2** (1968), no. 2, 127–145, See also [17].

[17] ______, *Semantics of context-free languages (correction)*, Mathematical Systems Theory **5** (1971), no. 1, 95–96.

[18] J. Kontos and J.C. Cavouras, *Knowledge acquisition from technical texts using attribute grammars*, The Computer Journal **31** (1988), no. 6, 525–530.

[19] John Kontos and George K. Papakonstantinou, *The interpretation of meta grammars describing syntax-directed interpreters using an attribute grammar interpreter*, no. 4, 435–436.

[20] Kai Koskimies, *A note on one-pass evaluation of attribute grammars*, BIT (1985), 439–450.

[21] C.H.A. Koster, *Syntax-directed parsing of* ALGOL *68 programs*, Proceedings Informal Conference on ALGOL 68 Implementation (Vancouver), University of British Columbia, pp. 61–69.

[22] ———, *Affix Grammars*, ALGOL 68 Implementation (Amsterdam) (J.E.L. Peck, ed.), North-Holland Publishing Company, pp. 95–109.

[23] C.H.A. Koster, *A technique for parsing ambiguous grammars*, GI – 4. Jahrestagung (Berlin) (D. Siefkes, ed.), Lecture Notes in Computer Science, vol. 26, Springer-Verlag, pp. 233–246.

[24] P. Kühling, *Affix-Grammatiken zur Beschreibung von Programmiersprachen*, Ph.D. thesis, Technische Universität Berlin, Fachbereich Informatik, February 1978, In German.

[25] H. Langeveld, EBLAUW*: Een experimentele transducergenerator*, Master's thesis, Katholieke Universiteit Nijmegen, March 1989, In Dutch.

[26] J.M.I.M. Leo, *On the complexity of topdown backtrack parsers*, Proc. of the NGI-SION Symp, Utrecht, The Netherlands, pp. 343–355.

[27] ———, *A general context free parsing algorithm running in linear time on every LR(k) grammar without using lookahead*, Proceedings SION Conference (Amsterdam), pp. 53–67.

[28] H. Meijer, *An implementation of Affix Grammars*, Semantics Directed Compiler Generation (N.D. Jones, ed.), Lecture Notes in Computer Science, vol. 94, Springer-Verlag, pp. 320–349.

[29] H. Meijer, *Programmar: A translator generator*, Ph.D. thesis, Katholieke Universiteit Nijmegen, 1986.

[30] H. Meijer, *The project on* EXTENDED AFFIX GRAMMARS *at nijmegen*, WAGA.

[31] Donn Robert Milton, *Syntactic specification and analysis with attributed grammars*, Tech. Report 304, Computer Sciences Department, University of Wisconsin-Madison, 1977.

[32] M.P.G. Moritz, *Description and analysis of static semantics by fixed point equations*, Ph.D. thesis, Katholieke Universiteit Nijmegen, 1989.

[33] J. R. Nestor, B. Mishra, W. L. Scherlis, and W. A. Wulf, *Extensions to attribute grammars*, Tech. Report TL 83-36, Tartan Laboratories Incorporated, April 1983.

[34] Timo Niemi, *Specification of a query language by the attribute method*, BIT **24** (1984), 171–186.

[35] H.R. Osborne, *Choosy affix evaluation for* EAG *parsers*, Master's thesis, Katholieke Universiteit Nijmegen, 1989.

[36] F.G. Pagan, *Semantic specification using two-level grammars: Blocks, procedures and parameters*, Computer Languages **4** (1979), 171–185.

[37] George K. Papakonstantinou, *An interpreter of attribute grammars and its application to waveform analysis*, no. 3, 279–283.

[38] Thomas J. Penello, *Very fast LR parsing*, ACM SIGPLAN Notices **21** (1986), no. 7, 145–151.

[39] E. Simon, *A new programming methodology using attribute grammars*, Acta Cybernetica **7** (1986), no. 4, 425–436.

[40] Michael Sonnenschein, *Global storage cells for attributes in an attribute grammar*, 397–420.

[41] H.H. Vogt, S.D. Swierstra, and M.F. Kuiper, *Higher order attribute grammars*, ACM SIGPLAN Notices **24** (1989), no. 7, 131–145.

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

[43] Dashing Yeh and Uwe Kastens, *Improvements of an incremental evaluation algorithm for ordered attribute grammars*, ACM SIGPLAN Notices **23** (1988), no. 12, 45–50.

[44] D.M. Yellin, *Attribute grammar inversion and source-to-source translation*, Lecture Notes in Computer Science, vol. 302, Springer-Verlag, 1988.

[45] D.M. Yellin and E.-M.M. Muechstein, *The automatic inversion of attribute grammars*, no. 5, 590–599.

[46] A.J. van Zwol, AGFL*'s revisited—a parser system with integrated lexicon*, Master's thesis, Katholieke Universiteit Nijmegen, 1990.

Index