Generalized LR Parsing in Haskell

João Fernandes

a30730@correio.ci.uminho.pt

Techn. Report DI-PURe-04.11.01

2004, October

PURe

Program Understanding and Re-engineering: Calculi and Applications
(Project POSI/ICHS/44304/2002)

Departamento de Informática da Universidade do Minho
Campus de Gualtar — Braga — Portugal

DI-PURe-04.11.01
Generalized LR Parsing in Haskell by João Fernandes

Abstract

Parser combinators elegantly and concisely model generalised LL parsers in a purely functional language. They nicely illustrate the concepts of higher-order functions, polymorphic functions and lazy evaluation. Indeed, parser combinators are often presented as a motivating example for functional programming. Generalised LL, however, has an important drawback: it does not handle (direct nor indirect) left recursive context-free grammars.

In a different context, the (non-functional) parsing community has been doing a considerable amount of work on generalised LR parsing. Such parsers handle virtually any context-free grammar. Surprisingly, little work has been done on generalised LR by the functional programming community ([9] is a good exception).

In this report, we present a concise and elegant implementation of an incremental generalised LR parser generator and interpreter in Haskell. For good computational complexity, such parsers rely heavily on lazy evaluation. Incremental evaluation is obtained via function memoisation.

An implementation of our generalised LR parser generator is available as the HaGLR tool. We assess the performance of this tool with some benchmark examples.

1 Motivation

The Generalised LR parsing algorithm was first introduced by Tomita [14] in the context of natural language processing. Several improvements have been proposed in the literature, among others to handle context-free grammars with *hidden* left-recursion [11, 7]. An improved algorithm has been implemented in scannerless form in the SGLR parser [3]. More recently, GLR capabilities have been added to Yacc-like tools, such as Bison.

The advantage of GLR over LR is compositionality. When adding rules to grammars, or taking the union of several grammars, the LR parsing algorithm will stumble into conflicts. These conflicts must be eliminated by massaging the augmented grammar before parsing can proceed. This massaging effort is a pain by itself, but even more so because of its effect on the associated semantic functionality. It precludes as-is reuse of AST processing components developed for the initial grammar[1]. GLR requires no such grammar changes. It tolerates conflicts, just forking off alternative parsers as necessary. These alternative parsers will either be killed off when they run into parse errors or merged when they converge to the same state. In the end, a single parser may survive, indicating a non-ambiguous global parse. When several parsers survive, additional disambiguation effort is needed to select a parse tree from the resulting parse forest. GLR's performance can be lower than LR's, but not too dramatically so [11].

In the context of Haskell, two general approaches to parsing are in vogue. One approach is offered by the Happy parser generator which produces bottom-up parsers in Haskell from grammar definitions, much like Yacc does for C. Like Yacc, Happy is resctricted to LALR parsing, thus lacking compositionality as explained above. The other approach is offered by several libraries of parser combinators. With these, top-down parsers can be constructed directly in Haskell. The main disadvantages of this approach, and of LL parsing in general, is that it fails to terminate on left-recursive rules. To eliminate left-recursion, the LL parser developer is forced to massage his grammar, which requires quite some effort, and makes the grammar less natural.

Given the above considerations, it is natural to long for GLR support in Haskell[2]. One approach would be to extend Happy from LALR to

[1] For a more in-depth analysis of the draw-backs of traditional parsing methods, we refer the reader to [4].

[2] In earlier work, one of the authors provides GLR support for Haskell in a not fully integrated fashion, c.q. by invoking an external GLR parser [5, 8].

GLR[3]. In this report we embark on a more challenging approach: to provide GLR support directly in Haskell in the form of parser combinators. Our solution extends our previous work on regular languages avaliable in the HaLeX library [12]. In brief, our solution will run as follows. We provide a datatype for context-free grammar representation, which the Haskell developer will use to define syntax (Section 3). We will provide functions that from these grammars construct deterministic and non-deterministic automata, which subsequently are turned into action tables which can be fed to a generalized LR acceptance function (Section 4). To faithfully model the GLR algorithm, including its state merging, we introduce function memoization at the appropriate place (this and other possible optimizations are described in Section 5). In this report, we consider only Tomita's original algorithm, allowing us to handle left-recursion, but not *hidden* left recursion. In future, we also hope to capture known improvements of Tomita's algorithm.

To compare our solution with other parsing approaches, we integrated our GLR parsing support into a stand-alone tool (Section 6), and we performed a series of benchmarks (Section 7).

2 Ambiguities in Languages

The popularity of Generalized Parsing is growing since it is able solve many problems that are common in other technologies based on LL and LR algorithms, and since it is able to handle both real programming languages and domain specific languages.

There are syntactic features in real programming languages (e.g., C,Haskell, JAVA, etc) that do not fit the restricted classes LL(K) or LALR(k) at all. The main reason this is that they are ambiguous in one way or the other.

Some grammars have simple conflicts that can be solved either by computing more look-ahead or by throwing alternative derivations in parallel. What Generalized Parsing offers is exactly this functionality. Other grammars, with some more serious ambiguities, may lead to all derivations being accepted as valid. As result of parsing, a collection of derivations (a parse forest) may be obtained instead of a single derivation (a parse tree).

Many of these more serious ambiguities are present in real programming languages. We will briefly analise ambiguities in two of these cases:

[3] During the development of this project, such an extension was pre-announced in the Haskell mailing list. Although, and still today, the tool is not publicly available.

the C programming language type definition ambiguity and the indentation ambiguity in Haskell.

- Type-definition in C
  The hardest problem that a C parser has to solve is to distinguish type names (introduced via a typedef) from variable names. This means that is it possible that certain identifiers can be parsed as either type identifiers or variable identifiers due to the overloading possibility of certain operators. Let's take a quick look at the following piece of C code:

  ```
  {Bool *b1;}
  ```

  This statement above can be interpreted in two different ways: the first would be to interpreter it as multiplying the Bool and b1 variables; the second as a declaration of a variable b1 to a Bool. The C compiler would always choose the first alternative unless Bool was declared earlier in the program to be a type using a typedef statement.

  Also associated with this kind of ambiguity, let's now examine the following piece of C code:

  ```
  (a)&(b)
  ```

  This code could again lead to the following ambiguity: it could be parsed as the bitwise-and of a and b (if a is the name of a variable), or as the type-cast of the expression &b to type a if a is the name of a type.

- Indentation in Haskell
  Some languages, specially functional programming ones, are designed to use indentation to indicate blocks of code. This or any other kind of line-by-line oriented position information is not expressible in any context-free grammar, but without it the syntax of such a language is ambiguous.

  To demonstrate this type of ambiguity, let's analyse the following pieces of code that would be written in any typical functional programming style:

```
a = b
    where b = d
          where d = 1
          c = 2
```

```
a = b
    where b = d
          where d = 1
                c = 2
```

In the left-hand piece of code, the `c` variable is meant to belong to the first `where` clause. Without layout interpretation, `c` could as well belong to the second `where` clause, as shown in the right-hand side piece of code.

For more details on these aspects, we refer the reader to [10] and [15].

These two examples, or more generally, the ambiguities ocurring in real Programming Languages, can be solved using context (or semantic) information.

There are other kinds of ambiguities that indeed can be solved at parse time. Consider, for example, the following context-free grammar that defines a simple example language for expressions.

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow i$$

Considering that i represents the class of integer numbers, we could try to parse the sentence $4 + 5 * 6$. There would be 2 different ways of doing it. We could either parse the sentence $(4 + 5) * 6$ or the sentence $4 + (5 * 6)$, which obviates the fact that the grammar is an ambiguous one.

Although this grammar is ambiguous it can be parsed as a non-ambiguous one, using standard techniques, if we provide the parser with the aditional (semantic) information: the priority and precedence rules of the $+$ and $*$ operators. With such information, traditional parser techniques (e.g., LALR(1)) and parser generators (e.g., YACC) can parse expressions deterministically. For example, the YACC parser generator has a special notation (presented next) to define the precedence and priority rules of operators.

```
%left  '+'
%left  '*'

%%

expr    : expr  '+'  expr
        | expr  '*'  expr
        | 'i'
        ;
```

So, the first two lines stand for the + and ∗ operators being left associative; the lines are listed in order of increasing precedence. This means that the ∗ operator has greater priority over the + operator. With such rules, the parsers produced by YACC will parse the example sentence correctly, producing as result a single solution.

In the context of GLR parsing, these rules are known as disambiguation rules. This rules are used to disambiguate the parsers results. The definition of a powerful disambiguation language is defined in [3].

What generalized parsing proposes to solve is the problem of working (also) with this ambiguous kind of grammars (that define in some cases, as seen before, real languages).

3 Representing Context-free Grammars

A Context-Free Grammar (CFG) G is a 4-tuple $G = (T, N, S, P)$ where T is the set of terminal symbols or vocabulary, N is the set of non-terminal symbols, S is the root (non-terminal) symbol, and, finally, P is the set of productions (or rewrite rules). Rather then using Haskell's predefined tuples, we introduce the following algebraic datatype for context-free grammars, called *Cfg*:

```
data Cfg t nt
  = Cfg {terminals :: [Symb t nt]
        , nonterminals :: [Symb t nt]
        , root   :: Symb t nt
        , prods :: [Prod t nt]}
```

Here we use Haskell built-in lists to model sets (of symbols). The data type *Cfg* is parameterised with both the types of grammar terminal and non-terminal symbols, so that we can define grammars where symbols can not only be strings, but also characters, integers, etc.

Productions have a name and consist in a list of grammar symbols. This name and symbols list are grouped in Haskell pairs.

```
type Prod t nt
  = (ProdName, [Symb t nt])
```

The predefined Haskell *String* type is used to represent production names.

```
type ProdName
  = String
```

A production is a list of terminal symbols (constructor *T*) and non-terminal symbols (constructor *NT*). To be able to produce (GLR) parsers for a given context-free grammar, we need to expand it with a new root symbol (constructor *Root*) and a terminal symbol (usually named Dollar, constructor *Dollar*). Thus, we include two constructors to explicitly model such symbols.

```
data Symb t nt
  = Dollar
  | Root
  | T t
  | NT nt
```

A production is defined as a list of grammar symbols with a name. Thus, the *left* and *right-hand* sides of a production are easily defined as the pre-defined *head* and *tail* functions. To make the notation of our grammars as similar to BNF as possible we define an infix operator $\longmapsto$ to denote the usual "*derives to*" operator:

```
lhs_prod = head
rhs_prod = tail
l ⟼ r = l : r
```

Using this data types and these functions, we are able to write context-free grammars in Haskell. For example, the language of arithmetic expressions with the operators addition and multiplication is written as:

$$
\begin{aligned}
expr = Cfg\ & [T\ \texttt{'i'}, T\ \texttt{'+'}, T\ \texttt{'*'}] \\
& [NT\ \texttt{'E'}] \\
& (NT\ \texttt{'E'}) \\
& [(\texttt{"add"}, NT\ \texttt{'E'} \longmapsto [NT\ \texttt{'E'}, T\ \texttt{'+'}, NT\ \texttt{'E'}]) \\
& , (\texttt{"mul"}, NT\ \texttt{'E'} \longmapsto [NT\ \texttt{'E'}, T\ \texttt{'*'}, NT\ \texttt{'E'}]) \\
& , (\texttt{"val"}, NT\ \texttt{'E'} \longmapsto [T\ \texttt{'i'}])]
\end{aligned}
$$

where the terminal symbol i represents the lexical class of integer numbers.

One of the possible ways to visualize this grammar would be as the graph shown in Figure 1.

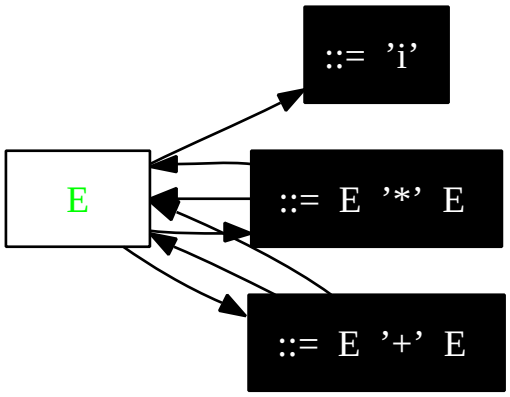

Fig. 1. Graphical Representation of the expr grammar

Having defined grammars in Haskell, the standard functions that test for grammar (or symbol) properties are elegantly and concisely written in this language. For example, the function that computes the *lookahead* of a production of a CFG is written in Haskell very much like its formal defintion:

$$
\begin{aligned}
& lookahead :: (Eq\ t, Eq\ nt) \Rightarrow Cfg\ t\ nt \rightarrow [Symb\ t\ nt] \rightarrow [Symb\ t\ nt] \\
& lookahead\ g\ p \\
& \mid nullable\ g\ [\,]\ (rhs\_prod\ p) = first\ g\ (rhs\_prod\ p) +\!\!+ follow\ g\ (lhs\_prod\ p) \\
& \mid otherwise \qquad\qquad\quad = first\ g\ (rhs\_prod\ p)
\end{aligned}
$$

where the functions *first* and *follow* compute the *first* and *follow* sets of a given symbol. The predicate *nullable* defines whether a sequence of symbols may derive the empty string or not. We omit here the definitions of these functions, since they also follow directly from their formal definition [1]. Several other functions are also easily expressed in Haskell (*e.g.*, testing for the LL(1) condition).

4 Generalised LR Parsing

This section presents the implementation of a Haskell-based Generalised LR (GLR) parser. Before we discuss the generalised LR parser, let us briefly recall the well-known LR parsing algorithm [1].

The LR parsing algorithms parse sentences following a *bottom-up* strategy. The derivation tree for a sentence f is built by starting from the leaves until the root is reached. This process can be seen as the *reduction* of the sentence f to the root of the grammar. This reduction process is performed as a sequence of reduction steps, where in each step a sub-sentence of f, that matches the right-hand side of a production p, is replaced by (or reduced to) the left-hand side non-terminal.

Let us consider our running example grammar. The sentence `12+5*6` is reduced to the root E through the following reducing steps:

`12+5*6`	reduce by production `val`
`E+5*6`	reduce by production `val`
`E+E*6`	reduce by production `add`
`E*6`	reduce by production `val`
`E*E`	reduce by production `mul`
`E`	accept

Although it is easy to perform the right sequence of reduction steps by-hand, we wish to define a function (*i.e.*, a LR parser) that performs the right reduction steps automatically. There are well-known techniques to construct such a LR parser. In Figure 2 we show the architecture of an LR parser (taken from [1]).

It consists of a generic LR acceptance function that is parameterised with the input sentence, the parsing tables (action and goto tables), and an internal stack. The parsing tables represent the grammar under consideration and they define a deterministic finite automaton (*goto table*) whose states contain the information needed to decide which actions to perform (*action table*). Before we explain the LR parsing algorithm, let us discuss how those tables are constructed.

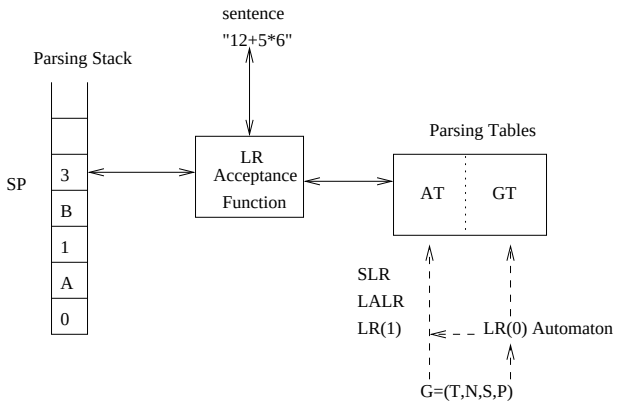

Fig. 2. The architecture of an LR parser

4.1 Representing Grammars as Automata

There are well-known techniques to represent context-free grammars as LR(0) automata. The idea is to define productions as *LR(0) items* and to use them as automaton states. A LR(0) item is a production with a *dot* in its right-hand side. Informally, it indicates how much of a production has been seen at a given point in the parsing process.

For example, production $E \to E + E$ induces the four LR(0) items

$$E \to .E + E, E \to E. + E, E \to E + .E, E \to E + E.$$

The last item is called a reducing item, since the dot is on its rightmost position. This means that all the symbols of the right-hand side of a production have been seen and, thus, they can be reduced to the left-hand side.

LR(0) items can be modelled in Haskell as pairs of productions and (dot) positions, as defined next:

type *Item sy* = ([*sy*], *Int*)

is_reducing_it (*p*, *i*) = *i* ≡ *length p*

The construction of the LR(0) automaton is performed in three steps: first the grammar is extended with a new root symbol, then the LR(0) items are induced, and, finally, a non-deterministic finite automaton is constructed. To construct the LR(0) automaton we use the support to model, manipulate and animate regular languages in Haskell, as provided in the HaLeX library [12]. Using this library, the construction is concise

and easily modelled in Haskell. Figure 3 shows the graphical representation (produced by HaLeX) of the Non-Deterministic Finite Automaton (NDFA) induced by our running example.

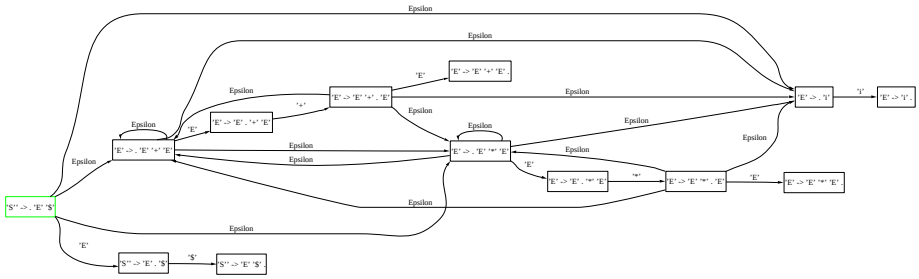

Fig. 3. Non-deterministic LR(0) Automaton.

The LR(0) automaton is obtained by simply converting this NDFA into a deterministic one. Figure 4 shows the LR(0) automaton.

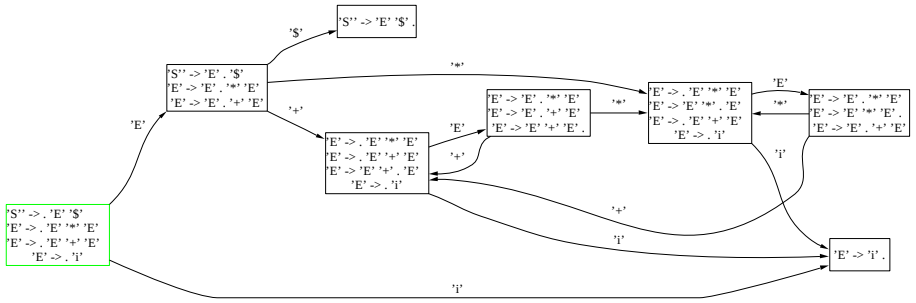

Fig. 4. Deterministic LR(0) Automaton.

The goto table used by the LR acceptance function is just a table representation of the transition function of this automaton.

4.2 From Automata to Action Tables

The LR acceptance function works as follows: with the state on the top of the stack and the input symbol, it consults the action table to determine which action to perform. Thus, we need to compute the actions to be performed in each state of the (deterministic) LR(0) automaton. There are four types of actions:

- *Shift*: the next state is shifted into the parsing stack.
- *Accept*: the parsing terminates with success.
- *Reduce by production p*: the parsing stack is reduced. The states corresponding to the symbols of the right-hand side of p are popped and the new state is pushed into the stack.
- *Error*: the parsing terminates with a parsing error.

To model these actions in Haskell we introduce a new data type, named *Actions*, with four constructor function, one per action.

```
data Action st pr = Shift st        -- Move to state st
                  | Accept          -- Accept the sentence
                  | Reduce pr       -- Reduce the stack
                  | Error           -- Error
```

The action table is defined as a list of rows, where each row consists of the DFA state (first column) and the remaining columns define the (unique) action to be performed for each symbol of the vocabulary.

```
type AT st pr = [(st, [Actions st pr])]
```

In the classical LR parsing algorithm, there is a *single action* in each entry of this table. In other words we say that there are no conflicts in the action table. A conflict occurs when there is more than one possible action in a state s for a particular symbol a. For example, if in s for symbol a we can do both a shift and a reduce action, then we say that there is *shift-reduce* conflict. In the case that we can reduce by two different productions we say that there is a *reduce-reduce* conflict. The LR algorithm handles conflict free grammars only.

There are several techniques to construct the action tables, namely, LR(0), SLR(1), LALR(1) and LR(1). They differ in terms of their complexity and power to express grammars. By power we mean the ability to construct conflict-free action tables. For example, the SLR - simple LR - is the easiest to implement, but the least powerful of the three. This means that it may induce non-conflict free action tables where the others do not.

4.3 The Generalised LR Algorithm

A generalised LR (GLR) parser, as originally defined by Tomita, starts the parsing process as a classical LR parser, but during parsing when

it encounters a shift-reduce or reduce-reduce conflict in the action table it forks the parser in as many parallel parsers as there are possibilities. These independently running simple parsers are fully determined by their parse stack and (equal) parsing tables.

In order to model GLR parsers in Haskell we start by defining the type of the action table. In the generalised version, the action table has to contain all possible (conflicting) actions to be performed on a state for a vocabulary symbol. Thus, each entry of the table consists of a *list of actions* and not by a single one (as defined previously).

type $AT\ st\ pr = [(st, [[Actions\ st\ pr]])]$

To construct the action table, we may use any of the techniques discussed previously, *i.e.*, LR(0), SLR(1), etc. The correctness of the GLR algorithm is not affected by the technique used, but only its performance. Note that fewer conflicts mean in this case fewer forks of the parser.

Next, we show the action table constructed for our expression grammar. It is easy to see that this grammar has four shift-reduce conflicts: the entries containing non-singleton lists.

	i	+	*	$
[S'→.E$,E→.E*E,E→.E+E,E→.i]	[Shift E→i.]	[Error]	[Error]	[Error]
[E→i.]	[Error]	[Reduce E→i]	[Reduce E→i]	[Reduce E→i]
[S'→E.$,E→E.*E,E→E.+E]	[Error]	[Shift E→.E*E,E→E+.E,E→.E+E,E→.i]	[Shift E→.E*E,E→E*.E,E→.E+E,E→.i]	[Accept]
[E→.E*E,E→.E+E,E→E+.E,E→.i]	[Shift E→i.]	[Error]	[Error]	[Error]
[E→.E*E,E→E*.E,E→.E+E,E→.i]	[Shift E→i.]	[Error]	[Error]	[Error]
[S'→E$.]	[Error]	[Error]	[Error]	[Error]
[E→E.*E,E→E+.E,E→E*E.]	[Error]	[Shift E→.E*E,E→.E+E,E→E+.E,E→.i ,Reduce E → E+E]	[Shift E→.E*E,E→E*.E,E→.E+E,E→.i ,Reduce E → E+E]	[Reduce E → E+E]
[E→E.*E,E→E*E.,E→E.+E]	[Error]	[Shift E→.E*E,E→.E+E,E→E+.E,E→.i ,Reduce E → E*E]	[Shift E→.E*E,E→E*.E,E→.E+E,E→.i ,Reduce E → E*E]	[Reduce E → E*E]

We are now in a position to model the GLR acceptance function in Haskell. The function *glraccept* gets as parameters the grammar and the input sentence. It returns a list with all possible (boolean) solutions as the result.

```
glraccept :: (Eq t, Ord t, Ord nt) ⇒ Cfg t nt → [t] → [Bool]
glraccept g inp = glr lookupTT lookupAT [State s] inp
  where
  eg@(Cfg t nt r p)    = expand_cfg g        -- The expanded grammar
  dfa@(Dfa v q s z d) = ecfg2LR_0Dfa eg     -- The LR_0 DFA

  at                   = e_gslr_at     eg   -- The Action Table

  lookupAT             = lookupAT t at      -- The lookup fun. for the AT
  lookupTT             = d                  -- The lookup fun. for the TT
```

This function works as follows: first, the grammar is extended with a new root symbol, and, then the LR(0) automaton is constructed. After that, the generalised action tables are constructed, and, finally the lookup function on the action and goto tables are defined.

The acceptance of the sentence is performed by function *glr*. As shown in Figure 2, this function is parameterised with the goto and action tables (*i.e.*, their lookup functions), the parser stack (starting the parsing process with the initial state of LR(0) on the stack) and the input sentence.

```
glr l_tt l_at sk [] = [ res
                       | ac ← acs
                       , res ← glr' l_tt l_at sk [] ac ]
  where st = top sk
        acs = l_at st Dollar


glr l_tt l_at sk inp@(h : t) = [ res
                               | ac ← acs
                               , res ← glr' l_tt l_at sk inp ac ]
  where st = top sk
        acs = l_at st (T h)
```

Here we see the distinctive behaviour of the GLR algorithm. The function *glr'* is invoked on all possible conflicting actions, not just on one.

The auxiliary function *glr'* models in Haskell the usual operations associated to each of the parser actions. Actually, this function is also used by the classical LR acceptance function (library function *lraccept*).

```
glr' l_tt l_at sk inp (Reduce pr) = glr l_tt l_at sk'' inp
  where sk'  = popN (sizeProd pr) sk   -- reducing the stack
        st   = top sk'                 -- acessing the state on its top
        st'  = l_tt st (lhs_prod pr)   -- moving to a new state
        sk'' = push st' sk'            -- pushing the new state

glr' l_tt l_at sk (h : t) (Shift st) = glr l_tt l_at sk' t
  where sk'  = push st sk

glr' l_tt l_at sk inp Accept = [True]
```

$glr'\ l\_tt\ l\_at\ sk\ inp\ Error = [False]$

Let us return to our running example and use our GLR parser to parse a simple expression:

```
Expr> glraccept expr (scanner "12+5*6+8")
[True,True,True,True,True]
```

Indeed, the result of the GLR parser is a list of solutions (all accepting the sentence in this case). Rather than just returning a list of boolean solutions, parsers usually construct an abstract syntax tree used for further analysis. A GLR parser produces a forest of such parse trees.

In order to produce more useful results, we have extended the GLR parser to produce XML trees - by using the HaXml library [16]- and ATerms [2]. Furthermore, we have also extended the HaXml library with a graphical representation of XML trees using the external tool **graphviz** - a graph vizualization system available from AT&T. For the example above, we would get the XML forest shown in figure 5.

As a result of building XML trees and ATerms, we can use the HaXml combinators or the Strafunski library [8] to perform transformations on such trees. For example, we may express disambiguation rules to select correct abstract trees from the resulting forest.

Regarding the example, the correct parse tree to select among the forest would be the one presented in figure 6, which respects the well known mathematical priority of operations.

4.4 Limitations

Tomita's algorithm, as implemented above has some limitations. The most important one is that it will fail to terminate on grammars that have hidden left-recursion. An example of such a grammar is the following:

$$
\begin{aligned}
hiden\_left\_recursive = Cfg\ & [T\ \texttt{'x'}, T\ \texttt{'b'}] \\
& [NT\ \texttt{'S'}, NT\ \texttt{'A'}] \\
& (NT\ \texttt{'S'}) \\
& [(\texttt{"p1"}, NT\ \texttt{'S'} \longmapsto [NT\ \texttt{'A'}, NT\ \texttt{'S'}, T\ \texttt{'b'}]) \\
& , (\texttt{"p2"}, NT\ \texttt{'S'} \longmapsto [T\ \texttt{'x'}]) \\
& , (\texttt{"p3"}, NT\ \texttt{'A'} \longmapsto [])]
\end{aligned}
$$

Bison's implementation of generalised LR parsing also fails to terminate on such grammars. For an explanation of these issues see [11, 7]. As

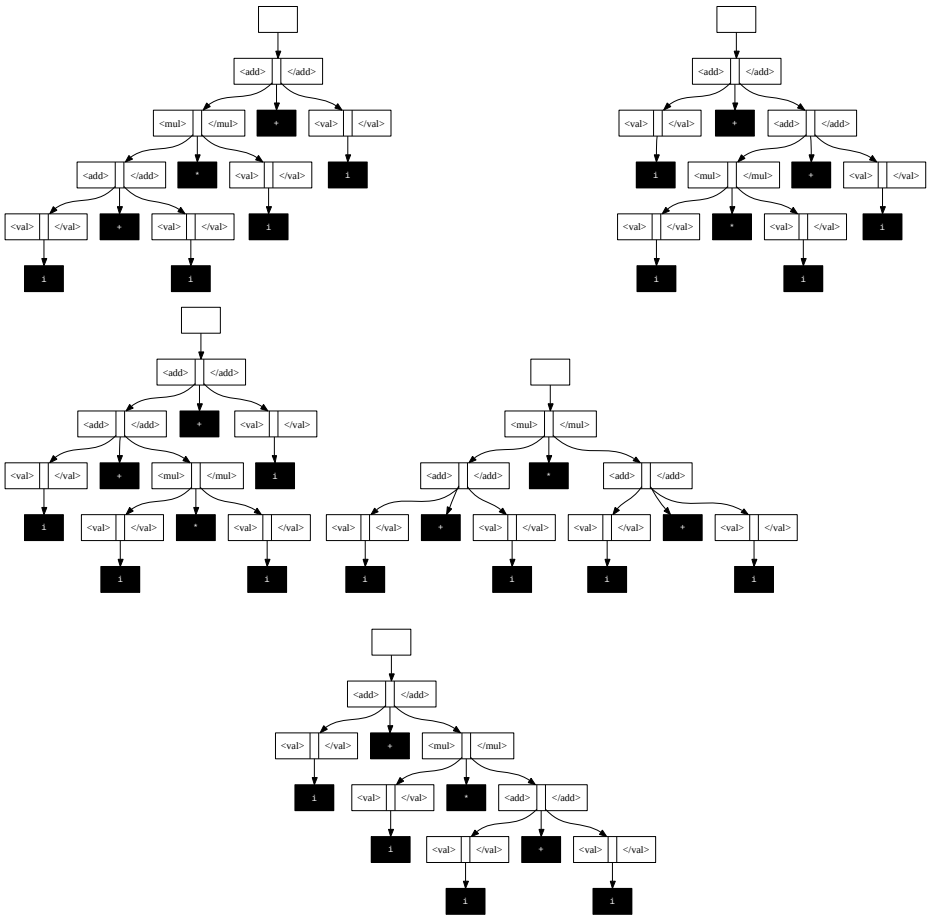

Fig. 5. The resulting parse forest

for Elkhound's implementation, it produces an error (not a parse one!) and aborts when trying to parse, for example, the sentence `xbbb`, which obviously belongs to the language defined by this grammar.

5 Optimisation

In this section we discuss two important optimisations of the GLR algorithm: the use of lazy evaluation not only to avoid the construction of all possible solutions, but also to schedule parallel parsers space efficiently, and the use of function memoisation to join two parsers.

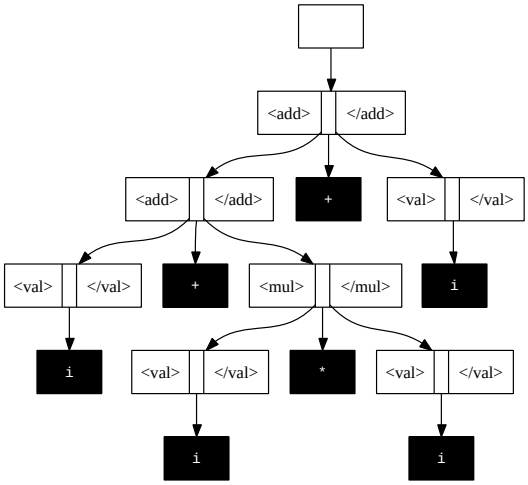

Fig. 6. The correct parse tree

5.1 Lazyness

Functional (top-down) parser combinators rely on lazy evaluation to avoid the construction of all the alternative solutions. If only a single parsing result is required, we can take the head of the list and all the other possible results are not computed at all. In GLR we can also rely in lazy evaluation to not compute all possible solutions (or parse trees), but only the ones we are interested in. For example, if we are interested in the first accepted solution we can define the following lazy GLR parser:

$$glracceptLazy :: (Eq\ t, Ord\ t, Ord\ nt) \Rightarrow Cfg\ t\ nt \rightarrow [t] \rightarrow Bool$$
$$glracceptLazy\ g\ inp = or\ \$\ glraccept\ g\ inp$$

5.2 Function Memoisation

Generalised LR splits the parser in other parsers when conflicts are encountered in the action table. These parsers then run in parallel; some may not accept the sentence while others may accept it. Some of these independent parsers, however, may converge exactly to the same parsing state after processing an ambiguous part of the input. In this case, an important optimisation would be to merge these parsers into a single one.

There are techniques to implement this optimisation in the context of imperative programming [11]. In functional programming, however, we can achieve this re-joining of parser functions by using standard function memoisation. Using this technique we memoise calls to the parser functions. Thus, if two parsers converge to the same parsing state, then the second parser to reach that state will find in the memo table the results of the previous one. In this case it will reuse such a result and will not compute it again.

Function memoisation can be implemented by using `ghc` primitive function *memo*. Thus, an incremental generalised LR parser is defined in Haskell as follows:

$$
\begin{aligned}
&\mathit{inc\_glr}\ \mathit{l\_tt}\ \mathit{l\_at}\ s\ \mathit{inp}@(h:t) = [\ \mathit{res} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{ac} \leftarrow \mathit{acs} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad , \mathit{res} \leftarrow \mathit{memo}\ (\mathit{glr'}\ \mathit{l\_tt}\ \mathit{l\_at})\ s\ \mathit{inp}\ \mathit{ac} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad] \\
&\quad \textbf{where}\ a = \mathit{top}\ s \\
&\qquad\quad\ \ \mathit{acs} = \mathit{l\_at}\ a\ (T\ h)
\end{aligned}
$$

6 The HaGLR library and tool

A library to provide support for GLR parsing in Haskell has been developed and it is available as part of the Haskell UMinho Libraries (HUL) at http://wiki.di.uminho.pt/documentation/. The API of this library is included in appendix A.

Having defined this GLR library, we can now construct useful tools to manipulate context-free grammars in Haskell. We have constructed two tools: a grammar interpreter and a parser generator. Both tools have the following features:

- they can use a standard LR or (incremental) GLR algorithm.
- they produce either ATerm or XML forests (trees) as result.
- they produce graphical representations of the LR(0) automata. Indeed the automata displayed in Figures 3 and 4 were automatically produced by our tools.
- they produce pretty-printing representation of the parse and action tables (see the table on page 12) by using the table pretty-printing algorithm included in the HUL [13].
- they produce a graphical representation of the XML trees. Actually the trees shown in Figure 5 were produced by our tools.

We have also developed a Web interface for the library. This interface provides all the funcionality described above and displays outputs in a graphical way.

6.1 Grammar interpreter

The grammar interpreter follows the architecture shown in figure 2, and is based on the function *glraccept* of page 12. Thus, given a grammar and a sentence, it computes the parsing tables and only after computing these tables the acceptance of the sentence (i.e., the parsing process) is performed.

As for using this grammar interpreter, it will only require to load parsing modules from the library into a Haskell interpreter such as GHCi. Afterwards, all the funcionality listed in appendix A will be available.

We can use the *glraccept* function to check whether the expression grammar accepts the sentence "i+i*i+i".

```
HaGlr> glraccept expr "i+i*i+i"
[True,True,True,True,True]
```

As expected there are five different parsing derivations, all of them conducting to an acceptance result.

6.2 Parser Generator and Parsing with generated parsers

The parser generator works in two phases, very much like any parser generator does: first, given a grammar, it computes the parse tables. Such parse tables are written in a file as a Haskell module. In the second phase, the parse tables (for a particular language) are compiled and linked to a previously defined main function.

Let's now describe in detail the usage of this tool.

– Parser Generator

Let's first explain the process of generating parse tables and parsers themselves.

We have produced a tool named **Table Generator**, that displays, when requested, the following help message:

```
Table Generator: a Haskell-based Parser Generator for HaGLR
version 0.0 (by Joao Saraiva, Joao Fernandes & Joost Visser)

Usage: TableGenerator [options] [file] ...

List of options:
 -g,-G FilePath .CFG Grammar         Parsing from a Cfg Productions List
 -d,-D FilePath .SDF Grammar         Parsing from a SDF Grammar definition
 -s,-S string   .Cfg starting symbol Specifying Grammar Starting Symbol
 -h,-?          .help                output a brief help message
```

So, this tool provides three alternatives. The first one is to use a compiled-in grammar, i.e., a grammar defined in the examples source of the library. To choose this way of building LR tables and parsers, just invoke the tool with no options, like:

```
$ tablegenerator
```

The second alternative is to feed the **Table Generator** with a Cfg grammar, as defined earlier. For simplicity purposes, it is actually enough to provide the list of productions and the starting symbol of the grammar. So, supposing Expr.cfg file contained the following list of productions:

```
[("plus", [NT "E",NT "E",T '+',NT "E"]),
 ("times",[NT "E",NT "E",T '*',NT "E"]),
 ("value",[NT "E",T 'i'])]
```

For table and parsers generation, in this case, just do:

```
$ tablegenerator -g Expr.cfg -s "E"
```

The last alternative specifies that the context-free grammar is written in the SDF notation. That is to say that **Table Generator** can be fed with an SDF grammar definition. Before explaining how to proceed with the table generation in this way, we will explain on how to build an SDF definition.

– The SDF formalism
  SDF is a formalism for the definition of syntax used for defining Context-Free Grammars. Indeed, there is available a large library of SDF grammars for many languages (e.g. JAVA, COBOL, BibTex). The idea behind the development of this front-end for HaGlr is to be able to use them and integrate real languages with our tool.

SDF is described in detail in [6] and this section is based on it.

An SDF definition consists of five sections and has the following overall structure:

1. **sorts**: *names of domains or non-terminlas to be used in the other sections of the specification*
2. **lexical syntax**: *the rules of the lexical syntax*
3. **contex-free syntax**: *the rules of the concrete and abstract syntax*
4. **priorities**: *definition of priority relations between rules of context-free syntax*
5. **variables**: *naming schemes for variables*

We introduce the most significant features of SDF by means of an example in which we define the lexical, concrete and abstract syntax of a simple programming language (see Figure 7). In the **sorts** section, six names are declared. These names can be interpreted in two ways:

- as non-terminlas of a lexical or a context-free grammar, and
- as names of the domains used to construct abstract syntax trees.

We will use this dual interpretation of sorts to achieve an automatic mapping between sentences and abstract syntax trees. In the lexical syntax section, we define a space, a tabulation, and a newline character as layout characters (line 3). In addition, the form of identifiers (line 4) and numeric constants (line 5) is defined.
In the **context-free syntax** section, the concrete *and* abstract syntax are defined:

- The concrete syntax is obtained by using the "non-terminal" interpretation of sorts and reading the rules from "right to left" as ordinary grammar rules.
- The abstract syntax is obtained by using the "domain" interpretation of sorts and reading the rules from left to right as definitions of (typed) constructor functions for abstract syntax trees. The sort names appearing in function definitions define the types of the arguments as well as of the result of these functions.

Some other features illustrated by the context-free syntax in Figure 7 are:

- Rules can define lists with or without separators (line 9).
- Rules may have a **bracket** attribute. Such rules are used only for grouping language constructs, but do not contribute to the abstract syntax (lines 8 and 17).

```
1.  sorts ID NAT PROGRAM STATEMENT SERIES EXP
2.  lexical syntax
3.    [ \t\n\r] -> LAYOUT
4.    [a-z] [a-z0-9]*                    -> ID
5.    [0-9]+                             -> NAT
6.  context-free syntax
7.    program SERIES                     -> PROGRAM
8.    begin SERIES end                   -> SERIES {bracket}
9.    { STATEMENT ";" }*                 -> SERIES
10.   ID ":=" EXP                        -> STATEMENT
11.   if EXP the SERIES else SERIES      -> STATEMENT
12.   until EXP do SERIES                -> STATEMENT
13.   EXP "+" EXP                        -> EXP {left}
14.   EXP "-" EXP                        -> EXP {non-assoc}
15.   EXP "*" EXP                        -> EXP {left}
16.   EXP "/" EXP                        -> EXP {non-assoc}
17.   "(" EXP ")"                        -> EXP {bracket}
18.   ID                                 -> EXP
19.   NAT                                -> EXP
20. priorities
21.   {left: EXP "*" EXP -> EXP, EXP "/" EXP -> EXP} >
      {left: EXP "+" EXP -> EXP, EXP "-" EXP -> EXP}
22. variables
23.   Exp                                -> EXP
24.   Series                             -> SERIES
```

Fig. 7. A simple programming language

- Rules may have various attributes defining their associativity properties (lines 13-16). We allow the definition of associative and left-, right-, or non-associatie operators.

In the **priorities** section, priority relations between rules in the context-free syntax are defined as well as the associativity of groups of different operators. As shown here, the operators $*$ and / have a higher priority than the operators $+$ and $-$.
Finally, in the **variables** section, naming schemes for variables are given. These variables can be used in two ways:

- as variables in semantics definitions added to the SDF definition,
- as "holes" in programs during syntax-directed editing.

The line numbers in the SDF definition are just for ease of reference in the text. They are not a part of the SDF definition proper.

In this way, and supposing Expr.def file contained the following SDF grammar definition, which defines the same toy expression language as the Cfg grammar given as example:

```
definition

module Main

exports
  sorts
  Expression
  context-free syntax
"i"                        -> Expression
Expression "+" Expression -> Expression
Expression "*" Expression -> Expression
```

Now, invoking:

```
$ tablegenerator -d Expr.def
```

would generate, as desired, the parsers and the needed tables.

This process works as follows: first, SDF grammar's character classes and layout are normalized by externally invoking a normalization command (SDF2 bundle needs to be instaled!). Then, the Cfg productions that correspond to the SDF productions are computed. After computing the list of Cfg productions, the complete grammar is calculated so that the table and parsing generation proceeds like if a Cfg grammar had been provided to **Table Generator**.

– Parsing with generated parsers

Let's now describe briefly how to use the generated parsers to parse sentences.

We have produced another tool for this purpose. It is called **HaGlr**, and is an Incremental and Lazy Haskell-based Generalised LR Parser. It would display the following help message, when required:

```
HaGlr: an Incremental and Lazy Haskell-based
       Generalised LR Parser

version 0.0 (by Joao Saraiva & Joao Fernandes)

Usage: HaGlr options [file] ...

List of options:
  -N,-n        .NDFA            Non-Deterministic LR_0 Automaton (graphviz)
```

```
-D,-d        .DFA             Deterministic LR_0 Automaton (graphviz)
-T,-t        .Action Table    SLR_1 Action Table
-G,-g        .GLR             Use Generalised LR Parsing
-C,-c        .Conflicts       Conflicts (SLR_1)
-M,-m        .Memoisation     Incremental Parsing
-S,-s        .Strict          "Strict" Parsing
-L,-l        .Lazy            Lazy Parsing
-X,-x        .Xml Semantics   Xml Tree building and visualization
-A,-a        .ATerm Semantics Aterm Tree building and its nodes counting
-I,-i string .input=string    specify input sentence
-o file      .output=file     specify output file
-h, -?       .help            output a brief help message
```

As we have seen, HaGlr is able to generate graphical representations of LR(0) automata, is able to pretty-print the parsing tables and to produce ASTs in XML or ATerm format. It is also able of computing SLR(1) conflicts and, of course, is yet able to parse sentences, and in various ways.

Parsing with "-G" or "-g" option will turn GLR parsing on; using the "-L" or "-l" parsing option will compute no solution but the first valid parsed one; "-M" and "-m" options will demand on parsers to memoise computations in the way described in Section 5.2. As for semantics, "-X" and "-x" options will build XML abstract syntax trees and "-A" or "-a" will build ATerm abstract syntax trees.

All the combinations of parser options are supported.

So, for example, to parse the sentence "i+i*i", which obviously belongs to the expression language, and using a lazy parser with XML tree builiding, we would just invoke (after generating the parsing tables in one of the ways described):

```
./glr -G -L -X -I"i+i*i"
```

The tool would say True, confirming that the sentence belongs to the language defined, and would provide the tree visualization of the solution found, which is presented in Figure 8.

6.3 HaGlr's WEB Interface

As mentioned, we have developed a WEB interface for the HaGlr tool. It is available online at the **Online demos** section of:

`http://wiki.di.uminho.pt/twiki/bin/view/PURe/PUReSoftware`

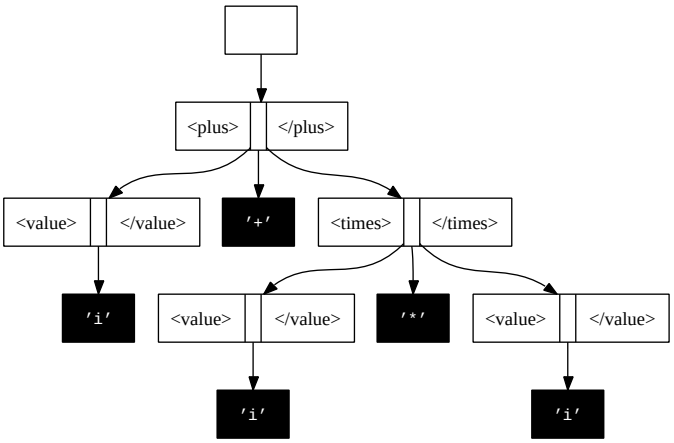

Fig. 8. XML AST resulting from parsing "i+i*i"

The point of any GUI is to make things easy on the user side. HaGlr's WEB interface is not an exception and is quite self-explanatory on its usage.

We will describe a possible interaction with this interface by running an example and presenting the graphical outputs produced.

When loading the interface, the user will be asked to provide a file containing a grammar definition. Once again, the grammar may be represented in this file by a list of Cfg productions or by an SDF definition.

In this example, we have chosen to load the expression grammar (in the list of Cfg productions representation), as can be seen in Figure 9.

Fig. 9. The WEB Interface initial menu

Upon loading, the interface displays a graphical representation of the loaded grammar (Figure 10). By asking for the grammar graph the user

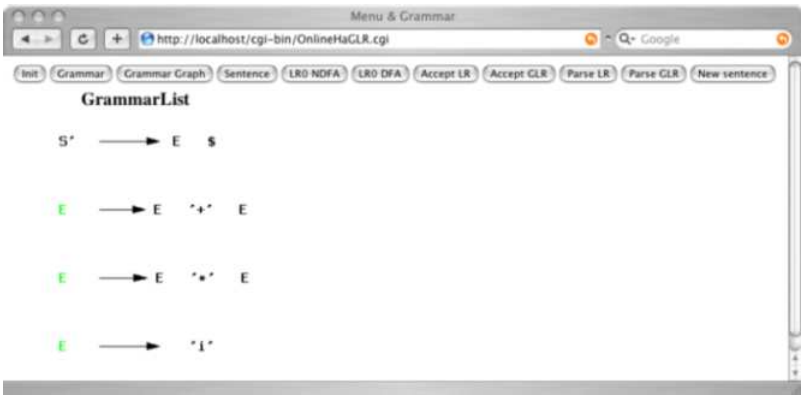

Fig. 10. The graphical representation of the expression grammar

would be shown, in this case, the graph presented in Figure 1.

It is also possible to view both the non-deterministic and deterministic LR(0) finite automata. For this example, the non-deterministic finite automaton would be like the automaton shown in Figure 3 and the deterministic one would be like the automaton shown in Figure 4.

The other functionalities provided by the interface are: test whether a sentence belongs to a language and parse sentences. In order to acess these functionalities, the user would first need to type an input sentence, in the way shown in Figure 11.

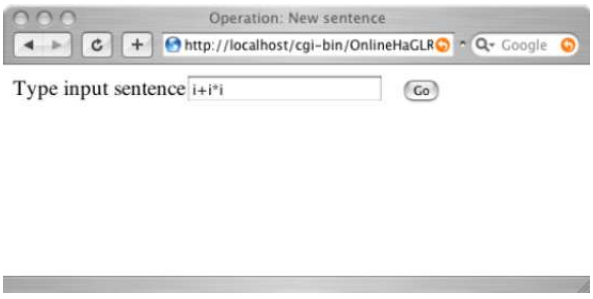

Fig. 11. The user interface menu for providing a sentence

Then, suposing he wanted to use GLR parsing, the user would be presented the parsing forest partially show in Figure 12, along with the acceptance information.

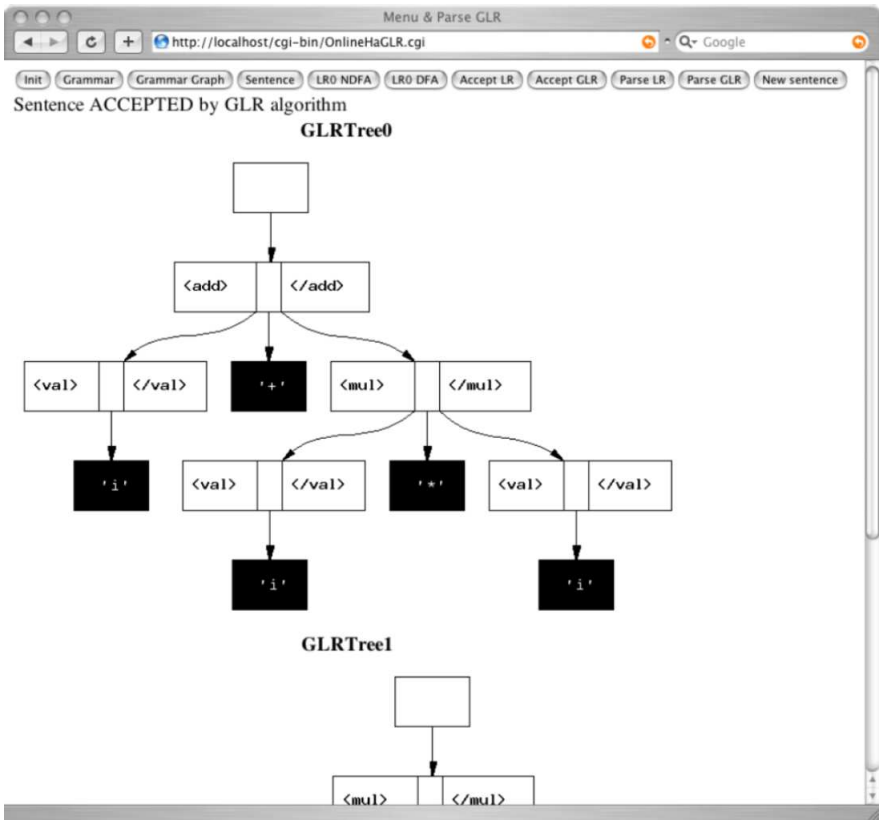

Fig. 12. The WEB Interface graphical displaying of the parse forrest

7 Benchmarking

7.1 Ambiguous Grammars

We have benchmarked our algorithm and other implementations of GLR parsing with three ambiguous grammars. Two of these grammars have been taken from the GLR literature [7]:

$$
\begin{aligned}
aj = Cfg\ & [T\ \texttt{'b'}] \\
& [NT\ \texttt{'S'}] \\
& (NT\ \texttt{'S'}) \\
& [(\texttt{"tresS"}, NT\ \texttt{'S'} \longmapsto [NT\ \texttt{'S'}, NT\ \texttt{'S'}, NT\ \texttt{'S'}]) \\
& ,(\texttt{"doisS"}, NT\ \texttt{'S'} \longmapsto [NT\ \texttt{'S'}, NT\ \texttt{'S'}]) \\
& ,(\texttt{"umb"}, \quad NT\ \texttt{'S'} \longmapsto [T\ \texttt{'b'}])]
\end{aligned}
$$

$$
\begin{aligned}
aj2 = Cfg\ & [T\ \texttt{'b'}] \\
& [NT\ \texttt{'S'}, NT\ \texttt{'T'}, NT\ \texttt{'A'}] \\
& (NT\ \texttt{'S'}) \\
& [(\texttt{"start"}, NT\ \texttt{'S'} \longmapsto [NT\ \texttt{'T'}]) \\
& ,(\texttt{"t1"}, \quad NT\ \texttt{'T'} \longmapsto [NT\ \texttt{'A'}, T\ \texttt{'b'}]) \\
& ,(\texttt{"t2"}, \quad NT\ \texttt{'T'} \longmapsto [NT\ \texttt{'T'}, NT\ \texttt{'T'}, NT\ \texttt{'T'}]) \\
& ,(\texttt{"a1"}, \quad NT\ \texttt{'A'} \longmapsto [NT\ \texttt{'T'}, T\ \texttt{'b'}, NT\ \texttt{'A'}, NT\ \texttt{'A'}, NT\ \texttt{'A'}]) \\
& ,(\texttt{"a2"}, \quad NT\ \texttt{'A'} \longmapsto [NT\ \texttt{'T'}, NT\ \texttt{'T'}, T\ \texttt{'b'}]) \\
& ,(\texttt{"a3"}, \quad NT\ \texttt{'A'} \longmapsto [])]
\end{aligned}
$$

Both of these grammars are highly ambiguous. As the input term gets longer, the number of ambiguities grows explosively. The third grammar contains only *local* ambiguities:

$$
\begin{aligned}
la = Cfg\ & [T\ \texttt{'a'}, T\ \texttt{'b'}] \\
& [NT\ \texttt{'S'}, NT\ \texttt{'A'}, NT\ \texttt{'C'}, NT\ \texttt{'T'}] \\
& (NT\ \texttt{'S'}) \\
& [(\texttt{"P0"}, NT\ \texttt{'S'} \longmapsto [NT\ \texttt{'T'}]), \\
& ,(\texttt{"P1"}, NT\ \texttt{'S'} \longmapsto [NT\ \texttt{'T'}, NT\ \texttt{'S'}]), \\
& ,(\texttt{"P2"}, NT\ \texttt{'T'} \longmapsto [NT\ \texttt{'A'}, T\ \texttt{'b'}, NT\ \texttt{'C'}]), \\
& ,(\texttt{"P3"}, NT\ \texttt{'A'} \longmapsto [T\ \texttt{'a'}]), \\
& ,(\texttt{"P4"}, NT\ \texttt{'C'} \longmapsto [NT\ \texttt{'A'}]), \\
& ,(\texttt{"P5"}, NT\ \texttt{'C'} \longmapsto [NT\ \texttt{'A'}, NT\ \texttt{'C'}])]
\end{aligned}
$$

The ambiguities of this grammar are local in the sense that all but a single parser will survive when the input string is accepted.

Time behaviour The results of parsing various input strings with these three grammars with various tools is shown in Figure 13. Four variations of our algorithm have been benchmarked. An M indicates that memoisation was turned on. An L indicates that only the first solution was computed. The other implementations we tested are Bison with GLR support, Elkhound, Happy with GLR support and SGLR.

command	aj	aj2	la
glr L	0.24	0.28	0.28
glr L M	0.36	1.32	1.96
glr	3.25	2.44	0.36
glr M	46.54	68.18	5.54
bison	1.24	16.12	0.11
elkhound	1.69	1.53	0.49
happy-glr	4.80	8.45	3.18
sglr	10.56	2.23	0.76

Fig. 13. Grammar aj, Input: bbbbbbbb, Grammar="aj2", input="bbbbbbbb", Grammar="la", input="abaaabaaaaaaaaabaaaaaaaaaaaaaaaabaaaaaabaabaaaaaaa"

When comparing the various variations of our implementation, we can make the following observations. Firstly, the running times for the variations where we only search for the first successful solution are significantly better than those where we search for all solutions. This demonstrates that the lazy evaluation delivers its promise of doing less work when less results are needed. Secondly, memoisation is not paying of in all tested cases. So, the amount of work that is being saved by function memoisation seems to be, for the work intensity requested in these cases, inferior to the amount of work that has to be done to implement memoisation.

Now, regarding our tool's best performanced variation and the other tools' performance, we can observe the following. In almost all cases, our runtimes are significantly better than the runtimes of all the others. Even the runtimes of our tool's second best performanced variation are, nearly half the times, better than the runtimes of the other tools.

Space behaviour The space behaviour of our implementation is vastly superior to those of some other implementations, namely Bison and SGLR. This is due to lazyness. Both Bison and SGLR can not parse input strings of more than 11 characters for the explosively ambiguous grammars. Beyond that number, the number of parallel parsers outgrows the default stack sizes of these tools. Of course, stack sizes can be increased to push the limits a little, but the non-linearity of the ambiguities renders this approach finite. Our implementation can handle input strings of many hundreds of characters, due to lazy evaluation. Not all parsers will be forked off at the same time, but only on demand. As for Elkhound and Happy, the space behaviour of these implementations is comparable with ours. Yet, since their time behaviour is significantly worst that our im-

plementation's, it's hard to determine exactly the input string size limit regarding space consumption for them!

Memoization In order to fully understand the runtimes of the memoized variations of out tool, we have produced some statistics on the number of times that two parsers get merged into a single one and on the number of times that we are trying to merge different parsers. We are using the same global ambiguous grammars and the same input strings we have used above.

	aj	aj2
Cache hits	71	111
Cache misses	86	700

Fig. 14. Number of sucessfully and unsuccesfully merged parsers

So, in both cases, there is a significant number of parsers being successfully merged. That is to say that, for ambiguous grammars, some forked parsers really end up converging into the same state, and that's when they are merged. For this examples, though, the effort of continuously keep track of the parsing work already done is actually higher than the effort that would be done repeatedly if memoisation wasn't used.

7.2 Non-Ambiguous Grammars

In order to benchmark the overhead of our GLR parser when processing non-ambiguous grammars, we consider the grammar of BibTeX: a language to define bibliographic databases. As input file we use the `AG.bib`, a large bibliographic database of attribute grammars related literature (compiled and available at INRIA). We compare the perfomance of our implementations with Happy. Both share the same scanner.

These results show that the Happy parser generator is faster than our LR and GLR implementations, for non-ambiguous grammars. Further optimisations have to be implemented in our library in order to approach Happy performance. Yet, the overhead of using GLR Happy parser generator over using LR Happy parser generator (runtimes become 3.5 times slower when using GLR support) seems to be greater than the overhead of using our GLR parser implementation over our LR parser implementation (runtimes become 2 times slower in the GLR implementation)!

	AG.bib		
	scanning	scanning+parsing	parsing
Happy	0.15	0.37	0.22
Happy with GLR support	0.15	0.91	0.76
LR	0.15	1.71	1.56
GLR	0.15	3.03	2.88

Fig. 15. Time in seconds.

7.3 The SDF Front-end

Now, we wanted to compare the performance of parsers generated via SDF front-end with the performance of parsers generated directly from Cfg grammars. So, we performed the following exercise: we had the toy expression grammar available in both SDF and Cfg representations; we just generated parsers for the expression language for both of the representations and tested their performance by repeting 1000 times the parsing of the sentence "i+i*i*i+i" for each of them. Results found are displayed in the next table.

command	Parsing via Cfg Grammar	Parsing via SDF front-end
glr L	2.46	2.64
glr L M	3.60	6.98
glr	2.71	4.38
glr M	10.43	26.59

Fig. 16. Input="i+i*i*i+i", time in seconds.

The results show that there is in fact some overhead resulting from working via SDF. As reasonable, the overhead is greater when more work is being requested, namely in the strict parser versions and in the memoised parser versions.

This results suggest that the Cfg grammar that is being computed from the SDF grammar is in fact "fatter" than it needs to be. This will lead to bigger action and go-to tables, which will lead to greater table generation and compilation times! Obviously, parsing times will also be affected by this facts.

This is more worrying if we recall that these results were obtained with tiny example grammars (recall Chapter 3 for the Cfg grammar and Chapter 6.2 for the SDF grammar).

The main interest of having developed this SDF front-end would be to use real language examples within the HaGlr tool. That was done next.

Firstly, we tried to parse sentences in the Java language. We picked up Java's SDF grammar definition and used as described the front-end in order to try to produce the needed parsing tables. Due to the dimension of this language and also to the overhead of using the SDF front-end, this process failed to terminate in the first 20 hours(!).

Given this situation, we decided to use a smaller (yet real) language. This time we used BibTeX language. Again we found the SDF grammar for this language and tried to generate the parsing tables for it. After normalization, this grammar has 80 SDF productions which should not be too many to work with. In fact, we managed to create the parsing tables and parsers into Haskell source code files (although this process took hours, which is beyond reasonable!). The problem was that, when trying to compile this source files in order to produce **HaGlr** tool, again we failed to terminate within the 24 hour bound.

Given so, we will have to work on one of the following situations, or maybe on both combined: the first possible improvment to be done would be to change the data-types in the generated Haskell source files (move List elements to Array elements, for example); the other possible improvment to work on would be to try to make each generated Cfg grammar thinner, in the sense that they would result in smaller parsing tables which will result in decreased compilation and parsing times.

Furthermore we can combine such approaches with well-known techniques to reduce the size of the parser tables: there are techniques to eliminate automaton states and as a result eliminate rows on those tables.

In order to determine the number of productions above which parsing generation and compilation times fall out of reasonable, we thought about extending the SDF's expression grammar more and more and see where this would leed us. The values we were getting are presented next.

What we can clearly see is that parsers and tables generation and compilation times are growing exponentially in the SDF productions of

Number of SDF Productions after normalization	Tables and Parsers Generation	Tables and Parsers Compilation
13	0.77	8.22
29	10.83	42.04
48	275.46	600.43
67	705.14	11353.72

Fig. 17. Time in seconds.

the normalized grammars. This is most obvious for the generation of the parsers and the tables.

In this way, and considering 3 hours (11300 seconds) the reasonability limit, trying to use the SDF front-end for grammars with more than 70 productions will stumble into huge generation and compilation times.

The extended expression grammar turned out to look like the grammar presented in appendix B.

With this grammar we could parse sentences like:

```
$glr -G -L -X -I" if (x=0) then (y <- z+1); else (y <- z-1);"
True
```

7.4 Conclusions

We have defined and publicly released a Haskell based tool which implements the GLR parsing algorithm. After benchmarking its performance we can say that this tool seems to be faster, for ambiguous grammars, when comparing it with other well known implementations. This is (as expected) more clear when we compute only the first solution rather than all of the solutions. As for non-ambiguous grammars, our implementations' performance is, according to the results obtained, worst than Happy's, for the (real) example benchmarked. Either way, we are still working on implementing performance optimizations so that the time behaviour ofour tool can still be lowered.

We also compared the space behaviour of our implementation with others'. The tests performed indicate that our tool's space behaviour is significantly better than the other implementations'. This is again more obvious in the case where we are interested only in the first solution rather than all. But also in the case where all solutions are computed, the lazy evaluation of our algorithm ensures that memory consumption remains within bounds.

As described, we are also releasing an SDF Front-end within our tool, which we have benchmarked. The results obtained show that this tool is yet in need of some studying and developing effort in order to become a real integration tool for real languages (available as SDF grammars).

8 Future work

We have presented an implementation of GLR parsing in Haskell. We have explained the role of lazy evaluation and function memoisation in our approach. We have developed a SDF front-end for HaGlr. We have run benchmarks to assess performance issues.

We intend to pursue various directions of future work:

- Keep implementing well-known LR optimizations [1], in order to increase the performance of our tool and by this to achieve other tools' performance for all kinds of grammars.
- Implement well-known improvements of the Tomita algorithm, in particular Rekers' approach to handle hidden left-recursion.
- Study and implement optimizations in the SDF front-end, so that this front-end can really be an integration interface, within HaGlr, for real languages, whose grammars are available in SDF.
- Define grammar combinators that include EBNF style operators, to provide the CFG writter with a modular and incremental way to implement languages.

A The Library's API

– *Context-Free Grammars*

-- Data Type synonym for the Right Hand Side of a production
type $RHS\ t\ nt = [Symb\ t\ nt]$

-- The "derives to" operator
$(|- >)$ $:: sy \rightarrow [sy] \rightarrow [sy]$

-- Computes a Dot Graph representation of a grammar
$cfg2dotGraph :: (Eq\ t, Eq\ nt, Show\ t, Show\ nt) \Rightarrow Cfg\ t\ nt \rightarrow String$

-- Computes a Dot List representation of a grammar
$cfg2dotList :: (Eq\ t, Eq\ nt, Show\ t, Show\ nt) \Rightarrow Cfg\ t\ nt \rightarrow String$

-- Selects the left-hand side of a production
$lhs\_prod :: [Symb\ t\ nt] \rightarrow Symb\ t\ nt$

-- Select the right-hand side of a production
$rhs\_prod :: [Symb\ t\ nt] \rightarrow RHS\ t\ nt$

-- Tests whether a symbol is a terminal symbol
$is\_terminal :: (Eq\ t, Eq\ nt) \Rightarrow Symb\ t\ nt \rightarrow Cfg\ t\ nt \rightarrow Bool$

-- Selects the productions that have a common non-terminal as left-hand side
$prods\_nt :: (Eq\ t, Eq\ nt) \Rightarrow Cfg\ t\ nt \rightarrow Symb\ t\ nt \rightarrow [[Symb\ t\ nt]]$

-- Selects the produtions RHSs that have a common non-terminal as left-hand side
$rhs\_nt :: (Eq\ t, Eq\ nt) \Rightarrow Cfg\ t\ nt \rightarrow Symb\ t\ nt \rightarrow [RHS\ t\ nt]$

-- Tests if a sequence is nullable
$nullable :: (Eq\ t, Eq\ nt) \Rightarrow Cfg\ t\ nt \rightarrow [Symb\ t\ nt] \rightarrow [Symb\ t\ nt] \rightarrow Bool$

-- Computes the first of a sequence of symbols
$first :: (Eq\ t, Eq\ nt) \Rightarrow Cfg\ t\ nt \rightarrow RHS\ t\ nt \rightarrow [Symb\ t\ nt]$

-- Computes the first of a non-terminal symbol
$first\_N :: (Eq\ nt, Eq\ t) \Rightarrow Cfg\ t\ nt \rightarrow Symb\ t\ nt \rightarrow [Symb\ t\ nt]$

-- Computes the follow of a non-terminal symbol
$follow :: (Eq\ t, Eq\ nt) \Rightarrow Cfg\ t\ nt \rightarrow Symb\ t\ nt \rightarrow [Symb\ t\ nt]$

-- Calculates the lookahead of a production
$lookahead :: (Eq\ t, Eq\ nt) \Rightarrow Cfg\ t\ nt \rightarrow [Symb\ t\ nt] \rightarrow [Symb\ t\ nt]$

-- Calculates the lookaheads of a non-terminal symbol
$lookaheads\_nt :: (Eq\ t, Eq\ nt) \Rightarrow Cfg\ t\ nt \rightarrow nt \rightarrow [[Symb\ t\ nt]]$

-- Testing a non-terminal for the ll_1 condition
$ll\_1\_nt :: (Eq\ nt, Eq\ t) \Rightarrow Cfg\ t\ nt \rightarrow Symb\ t\ nt \rightarrow Bool$

-- Testing the grammar for the ll_1 condition
$ll\_1 :: (Eq\ t, Eq\ nt) \Rightarrow Cfg\ t\ nt \rightarrow Bool$

-- Computing the complete Cfg grammar from its list of productions

$prods2cfg$ $:: (Eq\ t, Eq\ nt) \Rightarrow [Prod\ t\ nt] \rightarrow nt \rightarrow Cfg\ t\ nt$

– *LR Parsing*

-- Expands a Cfg with a new root symbol and a new production
$expand\_cfg$ $:: Cfg\ t\ nt \rightarrow Cfg\ t\ nt$

$cfg2ct$ $:: (Eq\ t, Eq\ nt) \Rightarrow Cfg\ t\ nt \rightarrow CT\ (Symb\ t\ nt)$

-- Computes the LR(0) Ndfa equivalent to a grammar
$cfg2LR\_0Ndfa :: (Eq\ t, Eq\ nt) \Rightarrow Cfg\ t\ nt \rightarrow Ndfa\ (Item\ (Symb\ t\ nt))\ (Symb\ t\ nt)$

-- Computes the LR(0) Dfa equivalent to a grammar
$cfg2LR\_0Dfa :: (Ord\ t, Ord\ nt) \Rightarrow Cfg\ t\ nt \rightarrow Dfa\ [Item\ (Symb\ t\ nt)]\ (Symb\ t\ nt)$

$saveLR\_0Ndfa :: (Show\ t, Show\ nt, Ord\ t, Ord\ nt, Eq\ t, Eq\ nt) \Rightarrow Cfg\ t\ nt \rightarrow [Char]$

$saveLR\_0Dfa :: (Show\ nt, Show\ t, Ord\ t, Ord\ nt, Eq\ t, Eq\ nt) \Rightarrow Cfg\ t\ nt \rightarrow [Char]$

-- Computes the SLR(1) action tables
slr_at $:: (Ord\ t, Ord\ nt) \Rightarrow Cfg\ t\ nt \rightarrow AT\ [Item\ (Symb\ t\ nt)]\ [Symb\ t\ nt]$

-- Pretty prints the slr action table
$slrPPTable$ $:: (Show\ t, Show\ nt, Ord\ t, Ord\ nt) \Rightarrow Cfg\ t\ nt \rightarrow [Char]$

-- Look up function for the action table
$lookupAT$ $:: (Eq\ t, Eq\ nt, Eq\ st) \Rightarrow [Symb\ t\ nt] \rightarrow AT\ st\ pr \rightarrow st \rightarrow$
$Symb\ t\ nt \rightarrow Action\ st\ pr$

-- LR acceptance function
$lraccept$ $:: (Ord\ t, Ord\ nt) \Rightarrow Cfg\ t\ nt \rightarrow [t] \rightarrow Bool$

-- LR acceptance function with XML tree building
lr_xml $:: (Ord\ t, Ord\ nt, Show\ t, Show\ nt) \Rightarrow Cfg\ t\ nt \rightarrow [t] \rightarrow (Bool, Element)$

-- LR acceptance function with ATerm building
lr_aterm $:: (Ord\ t, Ord\ nt, Show\ t, Show\ nt) \Rightarrow Cfg\ t\ nt \rightarrow [t] \rightarrow (Bool, ATerm)$

– GLR parsing

-- Computes SLR(1) action table
$gslr\_at$ $:: (Ord\ t, Ord\ nt) \Rightarrow Cfg\ t\ nt \rightarrow AT\ [Item\ (Symb\ t\ nt)]\ [Symb\ t\ nt]$

-- Pretty prints the gslr action table (ASCII)
$gslrPPTable :: (Show\ t, Show\ nt, Ord\ t, Ord\ nt) \Rightarrow Cfg\ t\ nt \rightarrow [Char]$

-- Pretty prints in a file the gslr action table
$gslrPPTable2File :: (Show\ t, Show\ nt, Ord\ t, Ord\ nt) \Rightarrow Cfg\ t\ nt \rightarrow FilePath \rightarrow IO\ ()$

-- Look up function for the action table
$glookupAT$ $:: (Eq\ a1, Eq\ a) \Rightarrow [a] \rightarrow [(a1, [a2])] \rightarrow a1 \rightarrow a \rightarrow a2$

-- GLR acceptance function
$glraccept$ $:: (Eq\ t, Ord\ t, Ord\ nt) \Rightarrow Cfg\ t\ nt \rightarrow [t] \rightarrow [Bool]$

-- incremental GLR acceptance function

$inc\_glraccept :: (Eq\ t, Eq\ nt, Ord\ t, Ord\ nt) \Rightarrow Cfg\ t\ nt \rightarrow [t] \rightarrow [Bool]$

-- GLR acceptance function with ATerm building
$glr\_aterm :: (Ord\ t, Ord\ nt, Show\ t, Show\ nt) \Rightarrow Cfg\ t\ nt \rightarrow [t] \rightarrow [(Bool, ATerm)]$

-- GLR acceptance function with XML tree building
$glr\_xml :: (Ord\ t, Ord\ nt, Show\ t, Show\ nt) \Rightarrow Cfg\ t\ nt \rightarrow [t] \rightarrow [(Bool, Element)]$

B Extended Expression SDF grammar

```
definition

module Main

exports
  sorts
         Expression
  context-free syntax
        Expression "+" Expression    -> Expression
        Expression "*" Expression    -> Expression
        Expression "-" Expression    -> Expression
        Expression "/" Expression    -> Expression
        "(" Expression ")"           -> Expression
        "[" Expression "]"           -> Expression
        "{" Expression "}"           -> Expression
        Int Sep Int                  -> Expression
        Int                          -> Expression
        I Int                        -> Int
        I                            -> Int
        Variable                     -> Expression
        Var                          -> Variable
        Var Variable                 -> Variable
        "if "     Expression
        "then "   Expression End
        "else "   Expression End     -> Expression
        "while "  Expression
        "do "     Expression End     -> Expression
        "for "    Expression
                  Expression End     -> Expression
        Expression Cond Expression   -> Expression
        Expression " <- " Expression -> Expression
        Comment Expression           -> Expression
```

```
      Expression Logic Expression  -> Expression
      Not Expression               -> Expression
      Expression "@" Expression    -> Expression
      "let "    Expression
      "where "  Expression         -> Expression

lexical syntax
      [\48-\57]                 -> I
      [a-zA-Z]                  -> Var
      [\44\46]                  -> Sep          %% , .
      [\60\61\62\171\187]       -> Cond         %% = > <
      [\59]                     -> End          %% ;
      [\35-\37]                 -> Comment      %% # $ %
      [\38\124]                 -> Logic        %% & |
      [\126]                    -> Not          %% ~
```

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, tools*. Addison-Wesley, 1986.
2. M. v. d. Brand, H. d. Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.
3. M. v. d. Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Compiler Construction (CC'02)*, Lecture Notes in Computer Science. Springer-Verlag, 2002.
4. M. v. d. Brand, M. P. A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117. IEEE, 1998.
5. M. de Jonge, T. Kuipers, and J. Visser. HASDF: A generalized lr-parser generator for haskell. Technical Report SEN-R9902, CWI, 1999. available at ftp://ftp.cwi.nl/pub/CWIreports/SEN/SEN-R9902.ps.Z.
6. P. K. J. Heering, P.R.H. Hendriks and J. Rekers. The syntax definition formalism sdf - reference manual. *Electronic Notes in Theoretical Computer Science*, 1992.
7. A. Johnstone, E. Scott, and G. Economopoulos. Generalised Parsing: Some Costs. In Evelyn Duesterwald, editor, *13th International Conference on Compiler Construction, CC/ETAPS'04*, volume 2985 of *LNCS*, pages 89–103. Springer-Verlag, April 2004.
8. R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, Jan. 2003.
9. P. Ljunglof. *Pure Functional Parsing, an advanced tutorial*. Thesis fot the Degree of Licenciate of Philosophy, Chalmers University of Technology and Goteborg University, 2002.
10. S. McPeak and G. C. Necula. Elkhound: A fast, practical glr parser generator. *Proceedings of Conference on Compiler Construction*, pages 73–88, 2004.

11. J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.
12. J. Saraiva. HaLeX: A Haskell Library to Model, Manipulate and Animate Regular Languages. In M. Hanus, S. Krishnamurthi and S. Thompson, editor, *Proceedings of the ACM Workshop on Functional and Declarative Programming in Education*, University of Kiel Technical Report 0210, pages 133–140, September 2002.
13. D. Swierstra, P. Azero, and J. Saraiva. Designing and Implementing Combinator Languages. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *Third Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, September 1999.
14. M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.
15. M. v. d. Brand, A. S. Klusener, J. Vinju, and L. Moonen. Generalized parsing and term rewriting: Semantics driven disambiguation. *Electronic Notes in Theoretical Computer Science*, 2003.
16. M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? *ACM SIGPLAN Notices*, 34(9):148–159, Sept. 1999. Proceedings of the International Conference on Functional Programming (ICFP'99), Paris, France.