Generalised LR Parsing in Haskell

João Fernandes, João Saraiva, and Joost Visser

Departamento de Informática
Universidade do Minho, Braga, Portugal

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, no work has been done on generalised LR by the functional programming community.
In this paper, 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 [10] 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 [9, 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

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

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 [9].

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 GLR. In this paper 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 [2]. 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 2). 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 3). To faithfully model the GLR algorithm, including its state merging, we introduce function memoization at the appropriate place. In this paper, 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 5), and we performed a series of benchmarks (Section 6).

2 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 sumbols or vocabulary, N is the 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*:

[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 [6, 8].

```
data Cfg sy   = Cfg { terminals :: [sy]
                    , nonterminals :: [sy]
                    , root :: sy
                    , prods :: [(ProdName,[sy])] }
type ProdName = String
```

Here we use Haskell built-in lists to model sets (of symbols). The data type *Cfg* is parameterised with the type of grammar symbols, so that we can define grammars where symbols can not only be strings, but also characters, integers, etc. 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 `|->` to denote the usual "*derives to*" operator:

```
lhs_prod = head
rhs_prod = tail
l |-> r = l : r
```

Using this data type 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:

```
expr = Cfg ['i','+','*']
           ['E']
           'E'
           [ ("add",'E' |-> ['E','+','E'])
           , ("mul",'E' |-> ['E','*','E'])
           , ("val",'E' |-> ['i'])
           ]
```

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

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

```
lookahead :: Eq sy => Cfg sy -> [sy] -> [sy]
lookahead g p | nullable g [] (rhs_prod p) = first g (rhs_prod p) ++
                                             follow g (lhs_prod p)
              | otherwise                  = first g (rhs_prod p)
```

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).

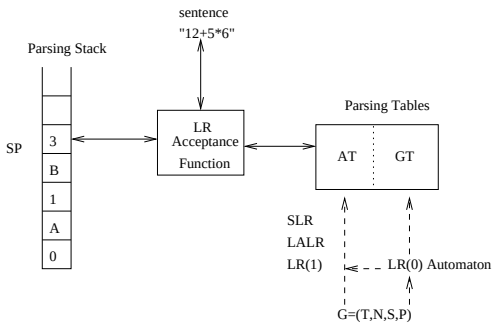

Fig. 1. The architecture of an LR parser

3 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 3 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.

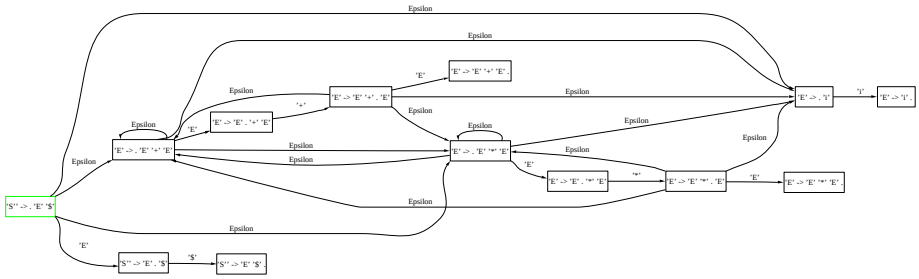

Fig. 2. Non-deterministic LR(0) Automaton.

3.1 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 [2]. Using this library, the construction is concise and easily modelled in Haskell. Figure 2 shows the graphical representation (produce by HaLeX) of the Non-Deterministic Finite Automaton (NDFA) induced by our running example.

The LR(0) automaton is obtained by simply converting this NDFA into a deterministic one. Figure 3 shows the LR(0) automaton.

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

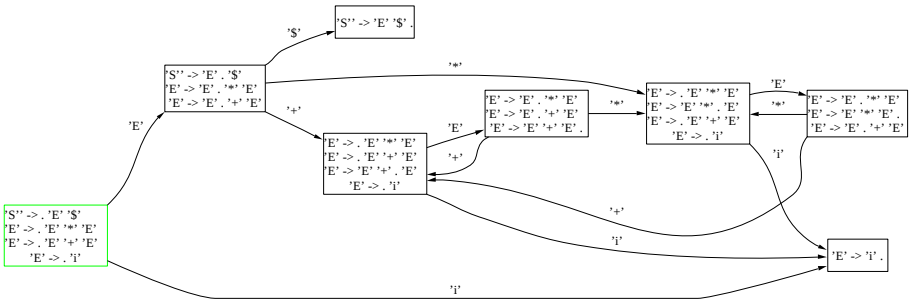

Fig. 3. Deterministic LR(0) Automaton.

3.2 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 input symbol and the next state are shifted into the parsing stack.
- *Accept*: the parsing terminates with success.
- *Reduce by production p*: the parsing stack is reduced. The symbols right-hand side of p and respective states are popped and the left-hand side 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 Actions pr = Shift
                | Accept
                | Reduce pr
                | 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 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 non-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.

3.3 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 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$,E→.E*E,E→.E+E,E→.i]	[Shift]	[Error]	[Error]	[Error]
[E→i.]	[Error]	[Reduce E→i]	[Reduce E→i]	[Reduce E→i]
[S'→E.$,E→E.*E,E→E.+E]	[Error]	[Shift]	[Shift]	[Accept]
[E→.E*E,E→.E+E,E→E+.E,E→.i]	[Shift]	[Error]	[Error]	[Error]
[E→.E*E,E→E*.E,E→.E+E,E→.i]	[Shift]	[Error]	[Error]	[Error]
[S'→E$.]	[Error]	[Error]	[Error]	[Error]
[E→E.*E,E→E+.E,E→E*E.]	[Error]	[Shift , Reduce E → E+E]	[Shift ,Reduce E → E+E]	[Reduce E → E+E]
[E→E.*E,E→E*E.,E→E.+E]	[Error]	[Shift ,Reduce E → E*E]	[Shift ,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 :: Ord sy => Cfg sy -> [sy] -> [Bool]
glraccept g inp =  glr lookupTT_g lookupAT_g [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_slr_at  eg      -- The Action Table

        lookupAT_g          = lookupAT t at     -- The lookup fun. for the AT
        lookupTT_g          = 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 3, 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 s []      = [ res
                          | ac <- acs
                          , res <- glr' l_tt l_at s [] ac
                          ]
   where (State a) = top s
         acs       = l_at a Dolar

glr l_tt l_at s inp@(h:t) = [ res
                            | ac <- acs
                            , res <- glr' l_tt l_at s inp ac
                            ]
  where  (State a) = top s
         acs = l_at a (Symb 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 `lraccpet`).

```
glr' l_tt l_at s inp (Reduce pr) = glr l_tt l_at s''' inp
  where s'          = popN (sizeProd pr * 2) s      -- reducing the stack
        (State a)   = top s'                        -- acessing the state on its top
        q'          = l_tt a (lhs_prod pr)          -- moving to a new state
        s''         = push (Gram (lhs_prod pr)) s'  -- pushing the lhs of the prod
        s'''        = push (State q') s''           -- pushing the new state

glr' l_tt l_at s inp@(h:t) Shift = glr l_tt l_at s'' t
    where (State a) = top s
          s'        = push (Gram (Symb h)) s
          q'        = l_tt a (Symb h)
          s''       = push (State q') s'

glr' l_tt l_at s inp Accept = [True]

glr' l_tt l_at s 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]
```

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 [11]- and ATerms [5]. As a result, 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.

3.4 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:

```
hiden_left_recursive = Cfg ['x','b'] ['S','A'] 'S'
                           [ ("p1", 'S' |-> "ASb")
                           , ("p2", 'S' |-> "x" )
                           , ("p3", 'A' |-> "" ) ]
```

Bison implementation of generalised LR parsing also fails to terminate on such grammars. For an explanation of these issues see [9, 7].

4 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.

4.1 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 :: Ord sy => Cfg sy -> [sy] -> Bool
glracceptLazy g inp = or $ glraccept g inp
```

4.2 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 [9]. 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:

```
inc_glr l_tt l_at s inp@(h:t) = [ res
                                | ac <- acs
                                , res <- memo (glr' l_tt l_at) s inp ac
                                ]
  where (State a) = top s
        acs = l_at a (Symb h)
```

5 The HaGLR library and tool

Having defined the HaGLR library, we can now define useful tools to manipulate context-free grammars in Haskell. We have constructed two tools: a parser generator and a grammar interpreter. Both tools take a CFG as input and calculate an LR table from it. The parser generator puts this LR table in a Haskell source file to be compiled into a stand-alone parser. The grammar interpreter uses the LR table directly. Both generated parsers and the grammar interpreter can perform either LR or (incremental) GLR parsing. In addition to parsing, these tools have the following functions:

- Generate graphical representation of LR(0) automata. See Figures 2 and 3.
- Pretty-print parse tables. See the table on page 7.
- Produce ASTs in XML or ATerm format.

The tools and the library are available from the authors websites.

6 Benchmarking

6.1 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]:

```
aj = Cfg ['b'] ['S'] 'S'
         [ ("tresS",'S' |-> "SSS")
         , ("doisS",'S' |-> "SS")
         , ("umb",'S' |-> "b")
         ]
```

```
aj2 = Cfg ['b'] ['S','T','A'] 'S'
          [ ("start",'S' |-> "T")
          , ("t1",'T' |-> "Ab")
          , ("t2",'T' |-> "TTT")
          , ("a1",'A' |-> "TbAAA")
          , ("a2",'A' |-> "TTb")
          , ("a3",'A' |-> "")
          ]
```

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:

```
la = Cfg ['a','b'] ['S','A','C','T'] 'S'
         [("P0",'S' |-> "T")
         ,("P1", 'S' |-> "TS")
         ,("P2", 'T' |-> "AbC")
         ,("P3",'A' |-> "a")
         ,("P4",'C' |-> "A")
         ,("P5", 'C' |-> "AC")]
```

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

command	aj	aj2	la
glr L	1.70	7.28	3.00
glr L M	0.40	1.65	1.12
glr	42.66	70.14	4.05
glr M	49.35	73.23	2.08
bison	0.07	0.06	0.11
sglr	8.05	1.14	0.78

Fig. 4. Time is seconds of 100 consecutive runs. Input sequences for the respective grammars are: "bbbbbbbb", "bbbbbbbb", and "abaaabaaaaaaaaabaaaaaaaaaaaaaaaabaaaaaabaabaaaaaaa".

Time behaviour The results of parsing various input strings with these three grammars with various tools is shown in Figure 4. 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, and SGLR.

The first observation is that both the Bison run times and the SGLR runtimes are significantly better than ours. Obviously, optimization work will need to be done to approach the time performance of these more mature implementations.

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 pays of in almost all cases.

Space behaviour The space behaviour of our implemenation is vastly superior to those of the other implementations. This is due to lazyness. Both Bison and SGLR can not parse input strings of more than 11 characters for the esxplosively ambiguous grammars. Beyond that number, the number 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.

6.2 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.

	AG.bib		
	scanning	scanning+parsing	parsing
Happy	0.15	0.18	0.03
Happy building Aterm	0.15	0.39	0.24
LR	0.15	00.32	0.17
LR building Aterm	0.15	18.93	18.78
GLR	0.15	01.24	01.09
GLR building Aterm	0.15	20.04	19.89

Fig. 5. Time in seconds.

These results show that the Happy parser generator is faster than our LR and GLR implementations. Further optimisations have to be implemented in our library in order to approach Happy performance.

6.3 Conclusions

From the benchmarks above we draw the following overall conclusion. The time behaviour of our Haskell implementation of GLR parsing is significantly worse than the other implementations. The only optimisation that we explored was the introduction of memoisation. In the conclusions we will list further oportunities for optimizations that may bring the time behaviour closer to the imperative implementations.

The space behaviour of our Haskell implementations is significantly better than the other implementations. This is most 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.

7 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 run benchmarks to assess performance issues.

We intend to pursue various directions of future work:

- Implement well-known LR optimizations [1].
- Develop an SDF front-end. This would allow us to use all grammars from the SDF grammar base with our Haskell based implementation.
- Implement well-known improvements of the Tomita algorith, in particular Rekers' approach to handle hidden left-recursion.
- Define grammar combinators that include EBNF style operators.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, tools*. Addison-Wesley, 1986.
2. J. ao 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.
3. M. G. J. 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. G. J. 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. v. d. Brand, H. d. Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000.
6. 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.
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. J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.
10. M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.
11. 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.