Faster Generalized LR Parsing

John Aycock and Nigel Horspool

Department of Computer Science,
University of Victoria,
Victoria, B. C., Canada V8W 3P6
{aycock,nigelh}@csc.uvic.ca

Abstract. Tomita devised a method of generalized LR (GLR) parsing to parse ambiguous grammars efficiently. A GLR parser uses linear-time LR parsing techniques as long as possible, falling back on more expensive general techniques when necessary.

Much research has addressed speeding up LR parsers. However, we argue that this previous work is not transferable to GLR parsers. Instead, we speed up LR parsers by building larger pushdown automata, trading space for time. A variant of the GLR algorithm then incorporates our faster LR parsers.

Our timings show that our new method for GLR parsing can parse highly ambiguous grammars significantly faster than a standard GLR parser.

1 Introduction

Generalized LR (GLR) parsing was developed by Tomita to parse natural languages efficiently [21]. Tomita observed that grammars for natural languages were mostly LR, with occasional ambiguities; the same can be said of C++ declaration syntax. Grammars for Graham-Glanville code generation are highly ambiguous.

Not surprisingly, parsers which deal strictly with unambiguous grammars can operate much faster than parsers for ambiguous grammars. This is crucial when one considers that the speed of input recognition is often highly visible to users. As a result, most artificial languages have unambiguous grammars by design, and much research has targeted speeding up parsers for unambiguous grammars. However, applications such as natural language understanding are rarely able to choose a convenient grammar, so there is still a need for fast parsers for ambiguous grammars.

Our work begins to address this problem. In this paper, we present an alternative method for constructing pushdown automata for use in LR parsing. We then show how these pushdown automata can be used to drive a GLR parser, giving a substantial speed increase.

2 LR and GLR Parsing

Space limitations prevent us from providing definitions for all notations and conventions used in this paper. Unless stated otherwise, we are using conventions similar to those used in compiler texts, such as [1].

Recall that a LR parser operates by "shifting" its input onto a stack, and "reducing" the stack when a handle is recognized on top of the stack. A handle is the right-hand side of a grammar rule, but *only* when reduction to the rule's left-hand side would correspond to a rightmost derivation step of the input [1].

Formally, if A $\rightarrow \alpha$ is a grammar rule and $S \overset{*}{\underset{rm}{\Longrightarrow}} \beta A w \underset{rm}{\Longrightarrow} \beta \alpha w$, then α is a handle at β. Under these circumstances, any prefix of $\beta\alpha$ is called a viable prefix. We use the term "viable string" to refer to $\beta\alpha$ in its entirety. (α and β symbolize strings of terminal and nonterminal symbols.)

Most current LR parsers are table-driven. They employ an automaton to find handles; this automaton's transitions and the parser actions are encoded into tables. A short generic algorithm is then sufficient to drive the LR parser.

GLR parsing builds on LR parsing. As we mentioned, Tomita observed that a number of ambiguous grammars were mostly LR. With that in mind, Tomita's algorithm behaves as a normal LR parser until it reaches a LR parser state where there is a conflict — the LR parser has a set of conflicting actions it could perform, and is unable to choose between them. A Tomita parser is not able to choose the correct action either, and instead simulates nondeterminism by doing a breadth-first search over all the possibilities [6].

Conceptually, one can think of the Tomita parser reaching a conflict, and starting up a new parser running in parallel for every possible action; each new parser "process" would have a copy of the original stack. A parser process that finds what seems to be erroneous input may assume that the action it took from the conflict point was the wrong one, and can terminate.

This cycle of a parser process starting others yields a wholly impractical algorithm. The time spent making copies of parser stacks could be enormous, not to mention the potentially exponential growth of the number of processes [23]. To address this, Tomita made two important optimizations:

1. A new process need not have a copy of its parent's stack. N processes can *share* a common prefix of a stack. From an implementation perspective, elements of the stack can all contain pointers to point to the previous element of the stack. Then, multiple stack elements can point to a common prefix.
2. There are a finite number of automaton states the parser can be in. Several processes may be in the same state, albeit they may have different stack contents. A set of processes that are in the same state can merge their stacks together, leaving one resulting process. This places an upper bound on the number of parsing processes that can exist.

   In a LR parser, its current state is the topmost state on the stack. So to merge N stacks, one would remove the top node from each — they must all have the same state number s — and create one node with state s that points to the remainder of the N stacks.

The result of these optimizations is called a graph-structured stack. (A slight misnomer, since the stacks actually form a directed acyclic graph.) The graph-structured stack in Fig. 1, for instance, corresponds to four processes and five conceptual stacks (the stack tops are the leaf nodes on the left-hand side).

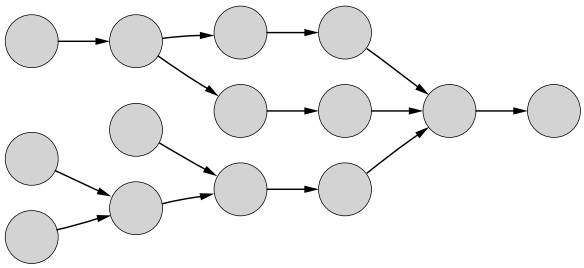

Fig. 1. A graph-structured stack

3 Faster LR Parsing

Much attention has been devoted to speeding up LR parsers, and the majority of this research pertains to implementation techniques. The argument is that interpreted, table-driven programs are inherently slower than hardcoded, directly-executable programs; given that, the best way to speed up a table-driven LR parser is to convert it into a directly-executable form that needs no tables.

[16, 8, 17, 3] all start with a LR parser's handle-finding automaton and translate it directly into source code — this source code can then be compiled[1] to create an executable LR parser. Basically, each state of the automaton is directly translated into source form using boilerplate code. This process tends to produce inefficient code, so these papers expend effort optimizing the source code output.

Several other papers [18, 19, 13, 14, 7] have taken a slightly different approach, introducing a technique called recursive ascent parsing. Here, a LR parser is implemented with a set of mutually recursive functions, one for each state[2] in a table-driven LR parser's handle-finding automaton.

Unfortunately, all of the above work is of limited use when applied to a GLR parser. LR parsers produce a *single* derivation for an input string. In terms of implementation, a LR parser only needs to keep track of a single set of information: the current parser state — what the parser is doing right now, and what it's done in the past. In a table-driven LR parser, this information is kept on an explicit stack; in a directly-executable LR parser, the information exists through a combination of the CPU's execution stack and program counter.

[1] Or assembled, as is the case in [16].

[2] Two functions per state are reputed to be required in [14].

In contrast, a GLR parser produces *all* derivations for an input string. This means that a GLR parser may need to keep track of multiple parser states concurrently. To construct a directly-executable GLR parser, one would need to maintain multiple CPU stacks and program counters. Certainly this is possible, but the overhead in doing so and switching between them frequently would be prohibitive, at least on a uniprocessor architecture.

Once direct execution of GLR parsers is ruled out, the obvious approach is to speed up table-driven LR (and thereby GLR) parsers. Looking at the LR parsing algorithm and its operation, one source of improvement would be to reduce the reliance on the stack. Fewer stack operations would mean less overhead, resulting in a faster parser.

The ideal situation, of course, is to have no stack at all! This would mean using finite automata to parse context-free languages, which is theoretically impossible [15]. Instead, we approximate the ideal situation. Our LR parsing method is an analogue to the GLR algorithm: it uses efficient finite automata as long as possible, falling back on the stack when necessary.

3.1 Limit Points

Our first step is to modify the grammar. When using a LR parser, the usual heuristic is to prefer left recursion in the grammar when possible; left recursion yields a shallow stack, because a handle is accumulated atop the stack and is reduced away immediately.

Non-left recursion may be ill-advised in regular LR parsers, but it is anathema to our method. For reasons discussed in the next section, we set "limit points" in the grammar where non-left recursion appears. A limit point is set by replacing a nonterminal A in the right-hand side of a grammar rule — a nonterminal causing recursion — with the terminal symbol $\perp_A$.

Figure 2 shows a simplified grammar for arithmetic expressions and the limit point that is set in it. The rules of the resulting grammar are numbered for later reference.

The process of finding limit points is admittedly not always straightforward. In general, there can be many places that limit points can be placed to break a cycle of recursion in a grammar. We will eventually be resorting to use of the stack when we reach a limit point during parsing, so it is important to try and find a solution which minimizes the number of limit points, both statically and dynamically.

For large grammars, it becomes difficult to select appropriate limit points by hand. The problem of finding limit points automatically can be modelled using the feedback arc set (FAS) problem [20]. Unfortunately, the FAS decision problem is NP-complete [9], and the corresponding optimization problem — finding the minimal FAS — is NP-hard [5]. There are, however, heuristic algorithms for the problem. We have used the algorithm from [4] due to its relative simplicity.

The number of limit points obtained for various programming language grammars is shown in Table 1. It is important to remember that these results were computed using a heuristic algorithm, and that the actual number of limit points

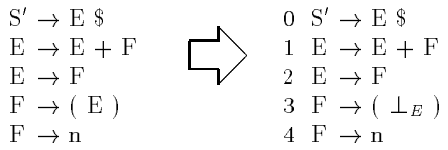

Fig. 2. Expression grammar and limit points

required may be lower. For example, starting with the computed limit points, hand experimentation revealed that no more than twelve limit points are needed for the Modula-2 grammar.

Table 1. Limit points derived heuristically

Ada	42
ANSI C	38
Java	23
Modula-2	23

3.2 Finite Automata

What we want to construct is a finite automaton which recognizes a viable string and remembers it. In other words, when a final automaton state is reached, the exact viable string is known. Simply recognizing a viable string with a finite automaton is unremarkable — standard LR parsers do this. The key point is being able to *remember* the viable string that was seen.

This means that the entire set of viable strings for a grammar must be enumerated, and a unique path must exist in our finite automata for each one. Unfortunately, while viable prefixes can be described by regular languages [11], most nontrivial grammars have an infinite number of viable prefixes, making enumeration of viable strings challenging.

This is where the limit points in the grammar come in. By choosing appropriate limit points, the set of viable strings for a grammar can be made finite and enumerable. Since viable strings can be generated by finding all paths through a LR parser's handle-finding automaton, this is the same as saying that the LR parser's automaton must have no cycles.

Once we have a finite set of viable strings, we build a finite automaton in three steps:

1. Construct a trie [12] from the viable strings, omitting any ϵ transitions. The trie structure ensures that each viable string has a unique path.

2. Add "reduction transitions," which indicate reduction by a particular grammar rule. Take all viable strings $\beta\alpha$, where α is a handle of the rule A $\rightarrow \alpha$. Let s be the start state; q_0 is the state at the end of the path $\beta\alpha$ starting with s; q_1 is the end state of the path βA, also starting with s. Assuming the rule A $\rightarrow \alpha$ is numbered k, add a transition from q_0 to q_1 labelled *reduce* k. As a special case, the final automaton state is the state at the end of the path $S\$$.
3. Delete transitions labelled with a nonterminal symbol, since these can never be read from an input string.

For example, the expression grammar in Fig. 2 has the set of viable strings $\{E\$, E+F, E+n, E+(\perp_E), F, n, (\perp_E)\}$. Its finite automaton is shown in Fig. 3. (We use a shaded circle to indicate the start state.)

3.3 Pushdown Automata

At this point, we have a finite automaton which only recognizes a subset of the original language. To remedy this, we add a stack and create a pushdown automaton.

How can a stack be incorporated? Intuitively, the $\perp$ transitions in the finite automaton are the natural places to push information onto a stack. When a $\perp$ transition appears, essentially the finite automaton is stating that it no longer has a sufficient number of states to remember any more. By pushing information at those points, a pushdown automaton is able to remember that which the finite automaton cannot.

To construct a pushdown automaton for a grammar G, we first build a finite automaton for G, FA_G, as described in the last section. Then, while there are $\perp$ transitions in FA_G, we perform the following steps:

1. Choose a transition $\perp_A$.
2. Create a new grammar $G_\perp$ from G. Initially, all rules in G are placed in $G_\perp$. Then set the start symbol for $G_\perp$ to be A, and remove all rules from $G_\perp$ that are unreachable from this new start symbol. Augment $G_\perp$ with the rule $A' \rightarrow A$ *pop*.
3. Construct a finite automaton for $G_\perp$ using the method in the last section; call it $\text{FA}_\perp$. $\text{FA}_\perp$ will act as a "subroutine" for FA_G in the sense that when FA_G reaches the $\perp_A$ transition, it will push a "return state" onto a stack, then go to $\text{FA}_\perp$'s start state. When $\text{FA}_\perp$ reaches a *pop* transition, it goes to a state which is popped off the stack.
4. Say that the transition on $\perp_A$ in FA_G was made from state q_0 to state q_1. Delete that transition from FA_G, replace it with a transition from q_0 to the start state of $\text{FA}_\perp$, and label the new transition *push* q_1.
5. Merge $\text{FA}_\perp$ into FA_G. Since these construction steps continue while there are $\perp$ symbols in FA_G, this means that all $\perp$ symbols in $\text{FA}_\perp$ eventually get replaced.

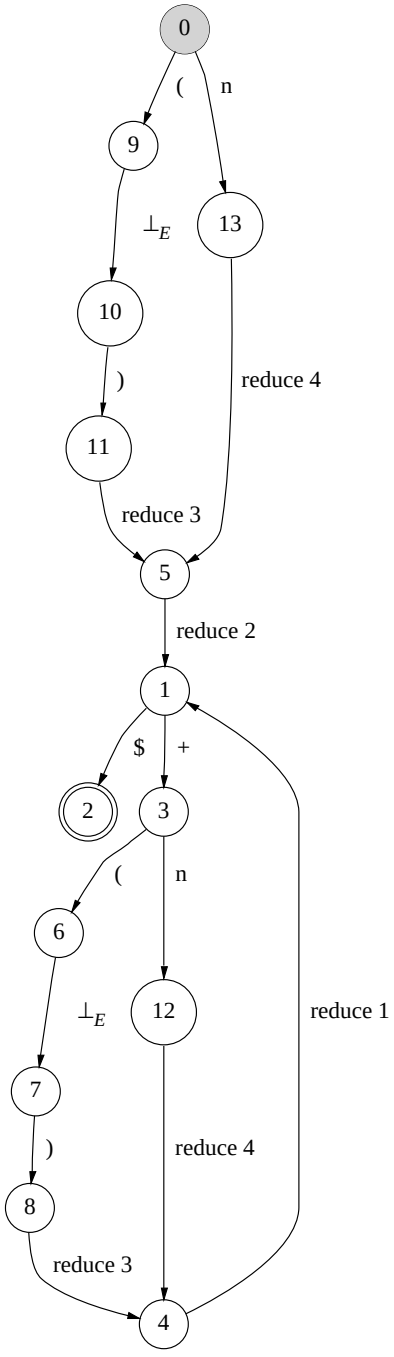

Fig. 3. Finite automaton for the expression grammar

The result of the above steps is a pushdown automaton for G; the pushdown automaton for our running example is shown in Fig. 4. As all the $\mathrm{FA}_\perp$ "subroutines" are built independently of any left context seen by their "caller," they can be re-used in other contexts. So the maximum number of $\mathrm{FA}_\perp$ that will be created for G is bounded by the number of limit points.

Figure 5 shows how the input string `((n + n))` is recognized by the pushdown automaton in Fig. 4. This example demonstrates that our pushdown automaton requires much fewer stack operations than a conventional LR parser.

4 Faster GLR Parsing

4.1 Algorithm

To use a pushdown automata from the last section as the engine for a GLR parser, we have devised a modified algorithm which is based on the work of Tomita [21–23].

Our algorithm uses two major types of structures: one for processes, the other for stack nodes.

1. Processes. Each process structure has a automaton state number and a pointer to a stack top associated with it.

   Process structures are linked into one of two lists. The current process list contains the processes that still require processing for the current input symbol; the pending process list contains processes that will need processing when the next input symbol is read. Every time a new input symbol is read, the pending process list becomes the current process list.
2. Stack nodes. There are two types of stack nodes:
   (a) Data nodes. This type of node contains the actual data of a process' stack. Each data node holds a single automaton state number, and a pointer to a previous stack node (i.e. pointing away from the stack top). If we used *only* this type of stack node, then we would have a tree-structured stack.
   (b) Fan-in nodes. These nodes are used to make the graph-structured stack; each one contains a set of pointers to previous stack nodes. When two process' stacks are merged, a fan-in node is created which holds pointers to both stacks. In our implementation, to bound the amount of effort required to find a data node, we add the constraint that a fan-in node may only point to data nodes.

The pseudocode for the modified GLR algorithm is shown in Figs. 6–7.

4.2 Results

We performed some timing experiments to compare a standard GLR parser with our modified GLR parser. As a basis for comparison, we used the public domain

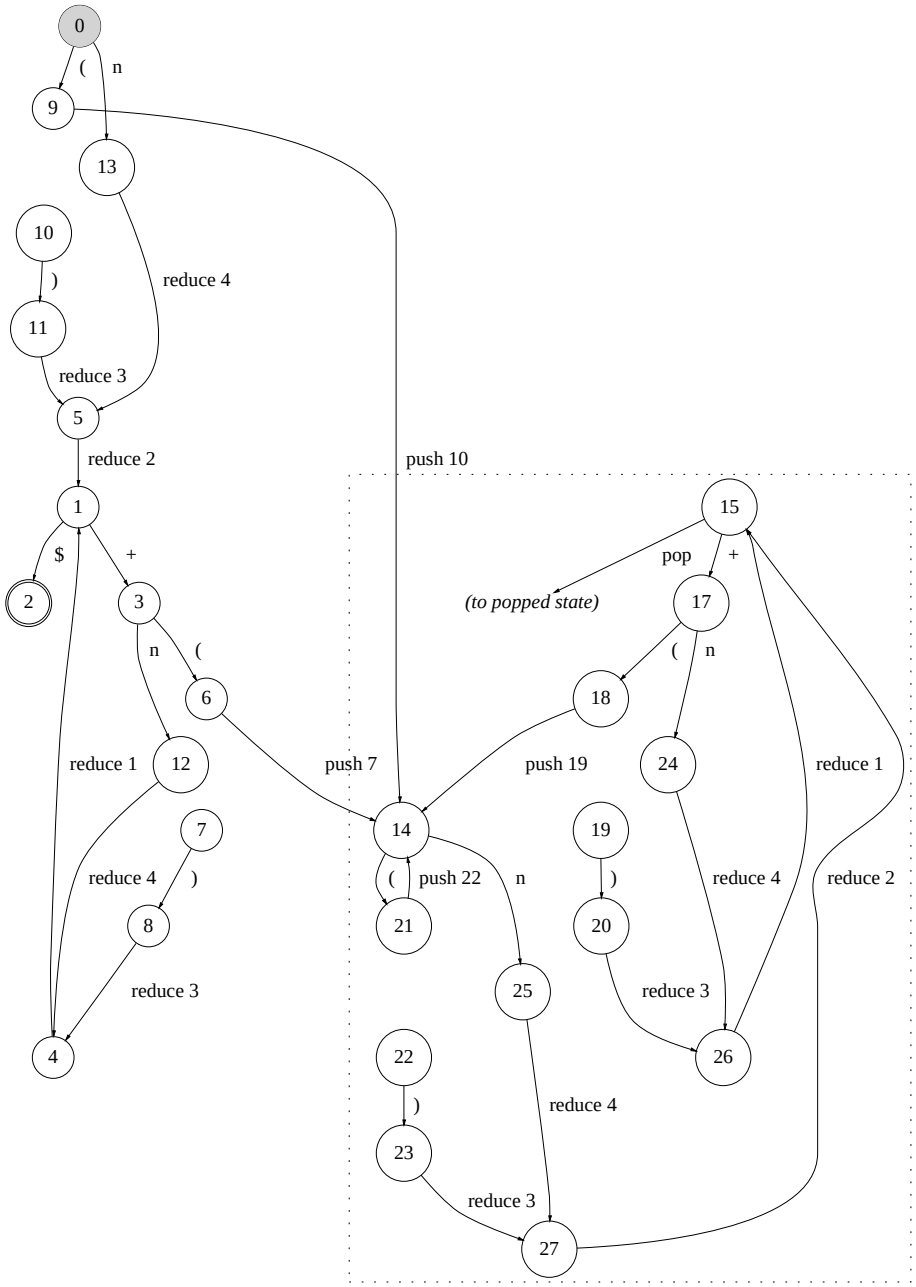

Fig. 4. Pushdown automaton for the expression grammar

Stack	State	Input	Action
	0	((n + n))$	shift 9
	9	(n + n))$	push 10, goto 14
10	14	(n + n))$	shift 21
10	21	n + n))$	push 22, goto 14
10 22	14	n + n))$	shift 25
10 22	25	+ n))$	reduce 4, goto 27
10 22	27	+ n))$	reduce 2, goto 15
10 22	15	+ n))$	shift 17
10 22	17	n))$	shift 24
10 22	24	))$	reduce 4, goto 26
10 22	26	))$	reduce 1, goto 15
10 22	15	))$	pop, goto popped state
10	22	))$	shift 23
10	23	)$	reduce 3, goto 27
10	27	)$	reduce 2, goto 15
10	15	)$	pop, goto popped state
	10	)$	shift 11
	11	$	reduce 3, goto 5
	5	$	reduce 2, goto 1
	1	$	shift 2, accept

Fig. 5. Example parser trace

GLR parser available from the `comp.compilers` Usenet newsgroup archive.[3] It uses LR(0) parse tables internally which are computed at startup. Both it and our modified GLR parser are implemented in C.

To ensure a fair comparison, we have modified our parser so that it incurs the same startup penalty and lexical analysis overhead as the public domain parser.

All tests were run on a Sun SPARCsystem 300 with 32M of RAM. Both parsers were compiled using `gcc` with compiler optimization (`-O`) enabled. To try and mitigate the effect of unpredictable system conditions on our timings, we ran the tests five times on each input; the results we report are the arithmetic mean of those times.

Our results are shown in Figs. 8–9 along with the grammars used, which we have numbered for convenience of reference. Each grammar is shown both with and without limit points.

Grammar 1 is an ambiguous grammar derived from one in [10]. Reductions in ambiguous grammars by rules with longer and longer right-hand sides are exponentially more expensive for GLR parsers. This is because GLR parsers, upon reduction by a rule $A \to \alpha$, must find all paths of length $|\alpha|$ from a stack top in the graph-structured stack. On the other hand, our modified GLR algorithm always takes a negligible time for reductions, as reflected in the results.

[3] `http://www.iecc.com` as of this writing.

```
function process(P, input) {
    foreach a ∈ action(input, P.state) {
        switch (a) {
        case SHIFT n:
            mergeInto(pending, n, P.stack)
        case REDUCE A → α, GOTO n:
            mergeInto(current, n, P.stack)
        case PUSH m, GOTO n:
            mergeInto(current, n, push(m, P.stack))
        case POP:
            let S be the set of stack data nodes atop P.stack
            foreach node (state, stack) ∈ S {
                mergeInto(current, state, stack)
            }
        }
    }
}

initialize pending process list to be empty
initialize current process list to be a single process,
    at the automaton's start state with an empty stack

while (current process list is nonempty) {
    input = getNextInputSymbol()
    while (current process list is nonempty) {
        remove a process P from the list
        process(P, input)
    }
    exchange the current and pending process lists
    if (input == EOF) {
        if (process in current process list is in accept state)
            accept input
        else
            reject input
    }
}
reject input
```

Fig. 6. Faster GLR parsing algorithm

```
function mergeInto(list, state, stack) {
    Looks in the specified process list for a process with
    a matching state as that passed in.  If it finds such
    a process, it simply merges its stack with the one
    passed in; if not, it creates a new process structure
    with the given state number and stack pointer, and adds
    it to the specified process list.
}

function push(state, stack) {
    Returns a new stack data node containing the given
    state and stack pointer.
}

function action(inputSymbol, state) {
    Based on its parameters, returns a set containing
    zero or more of:
        SHIFT n
        REDUCE A → α, GOTO n
        PUSH m, GOTO n
        POP
}
```

Fig. 7. Faster GLR parsing algorithm (continued)

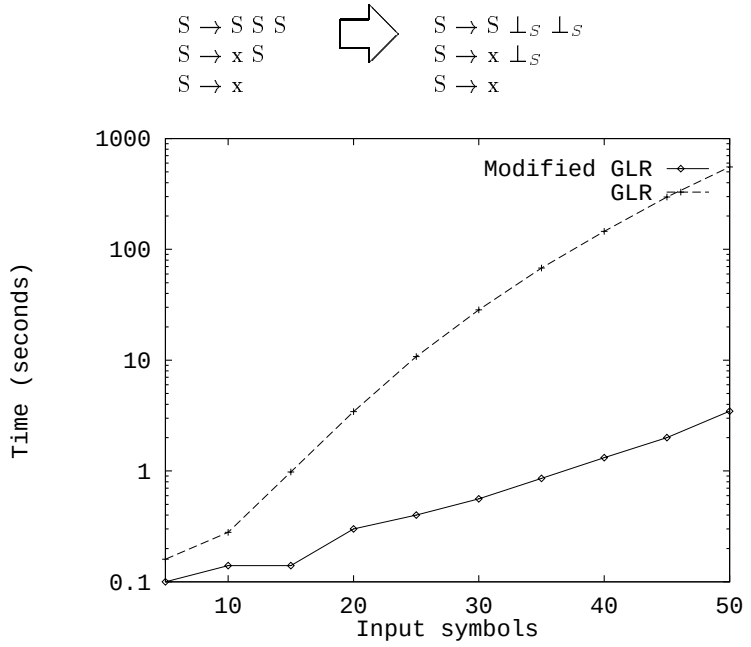

Fig. 8. Timings for Grammar 1

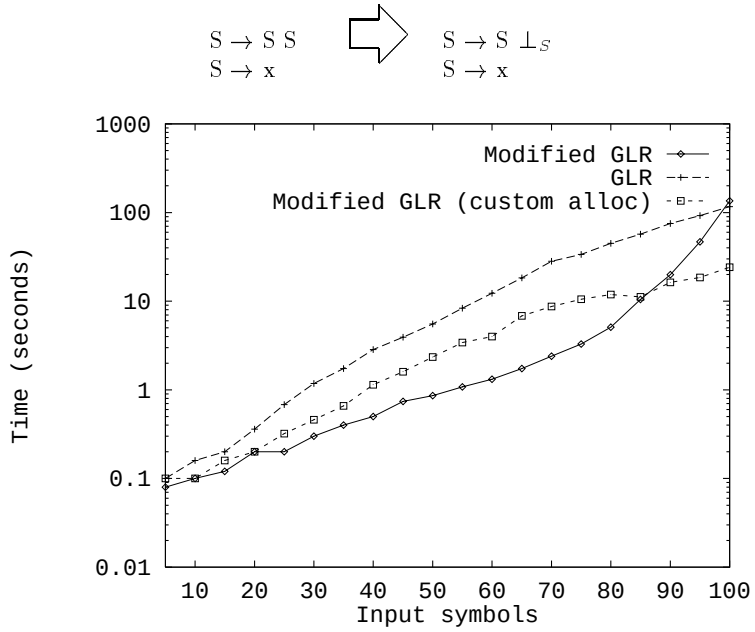

Fig. 9. Timings for Grammar 2

Grammar 2 is another ambiguous grammar from [10]. It is one of the worst cases for our modified GLR algorithm, requiring it to perform numerous stack operations on multiple stacks. This test is also interesting because it underscores the importance of memory management in GLR parsers. Profiling of our parser has shown that over 40% of total run time can be spent doing memory allocation and deallocation when parsing ambiguous grammars. Figure 9 shows our parser having an adversarial relationship with the standard C memory allocator, and the result of adding a custom-built memory allocator.

5 Future Work and Conclusion

There are a number of avenues for further work. Our GLR algorithm should be extended to take lookahead into account, and semantic actions should be supported. In terms of the grammar, the notion of limit points can be generalized so that recursion in the grammar is "unrolled" much as an optimizing compiler might unroll a loop; we have done some preliminary work on this possibility [2]. We would also like to conduct more experiments against other GLR parser implementations, to determine if the results we have obtained are typical.

In this paper, we have presented an alternative way to construct pushdown automata for use in LR parsers. These automata, when used in our modified GLR parsing algorithm, have substantially lowered parsing time when compared to a

regular GLR parser. Our timings show an improvement by up to a factor of ten for highly-ambiguous grammars.

By trading space for time — a larger LR parser in exchange for faster execution times — we are able to build GLR parsers which are faster and better suited to more widespread application outside the natural language domain.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. J. Aycock. *Faster Tomita Parsing*. MSc thesis, University of Victoria, 1998.
3. A. Bhamidipaty and T. A. Proebsting. Very Fast YACC-Compatible Parsers (For Very Little Effort). Technical Report TR 95–09, Department of Computer Science, University of Arizona, 1995.
4. P. Eades, X. Lin, and W. F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47:319–323, 1993.
5. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
6. D. Grune and C. J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Ellis Horwood, 1990.
7. R. N. Horspool. Recursive Ascent-Descent Parsing. *Journal of Computer Languages*, 18(1):1–16, 1993.
8. R. N. Horspool and M. Whitney. Even Faster LR Parsing. *Software, Practice and Experience*, 20(6):515–535, 1990.
9. R. M. Karp. Reducibility Among Combinatorial Problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Calculations*, pages 85–103. Plenum Press, 1972.
10. J. R. Kipps. GLR Parsing in Time $O(n^3)$. In Tomita [24], pages 43–59.
11. D. E. Knuth. On the Translation of Languages from Left to Right. *Information and Control*, 8:607–639, 1965.
12. D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.
13. F. E. J. Kruseman Aretz. On a Recursive Ascent Parser. *Information Processing Letters*, 29:201–206, 1988.
14. R. Leermakers. Recursive ascent parsing: from Earley to Marcus. *Theoretical Computer Science*, 104:299–312, 1992.
15. H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, 1981.
16. T. J. Pennello. Very Fast LR Parsing. In *Proceedings SIGPLAN '86 Symposium on Compiler Construction*, volume 21(7) of *ACM SIGPLAN Notices*, pages 145–151, 1986.
17. P. Pfahler. Optimizing Directly Executable LR Parsers. In *Compiler Compilers, Third International Workshop, CC '90*, pages 179–192. Springer-Verlag, 1990.
18. G. H. Roberts. Recursive Ascent: An LR Analog to Recursive Descent. *ACM SIGPLAN Notices*, 23(8):23–29, 1988.
19. G. H. Roberts. Another Note on Recursive Ascent. *Information Processing Letters*, 32:263–266, 1989.
20. E. Speckenmeyer. On Feedback Problems in Digraphs. In *Graph-Theoretic Concepts in Computer Science*, pages 218–231. Springer-Verlag, 1989.

21. M. Tomita. *An Efficient Context-Free Parsing Algorithm for Natural Languages and Its Applications*. PhD thesis, Carnegie-Mellon University, 1985.
22. M. Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic, 1986.
23. M. Tomita. An Efficient Augmented-Context-Free Parsing Algorithm. *Computational Linguistics*, 13(1–2):31–46, 1987.
24. M. Tomita, editor. *Generalized LR Parsing*. Kluwer Academic, 1991.