Even Faster Generalized LR Parsing

John Aycock[1], Nigel Horspool[1], Jan Janoušek[2], and Bořivoj Melichar[2]

[1] Department of Computer Science, University of Victoria,
Victoria, B.C., Canada V8W 3P6
{aycock|nigelh}@csc.uvic.ca
[2] Department of Computer Science and Engineering,
Czech Technical University, Karlovo nám. 13, 121 35, Prague, Czech Republic
{janousej|melichar}@cs.felk.cvut.cz

Abstract. We prove a property of generalized LR (GLR) parsing — if the grammar is without right and hidden left recursions, then the number of consecutive reductions between the shifts of two adjacent symbols cannot be greater than a constant. Further, we show that this property can be used for constructing an optimized version of our GLR parser. Compared with a standard GLR parser, our optimized parser reads one symbol on every transition and performs significantly fewer stack operations. Our timings show that, especially for highly ambiguous grammars, our parser is significantly faster than a standard GLR parser.

1 Introduction

Generalized LR (GLR) parsing is one of the fundamental parsing methods for context-free grammars (CFGs), and is used both in Computer Science and in Computational Linguistics. Information on GLR parsing can be found in [25], [26], and [20]. In [4] and [3] we presented a new way to construct GLR parsers which reduced the number of stack operations performed during parsing; timings showed that this modification can give a significant speed improvement over the standard (Tomita) GLR parser [25].

As suggested by their name, GLR parsers are based upon LR parsers, a commonly-used type of deterministic parser which can be used with a subset of CFGs. Standard LR parsers read their input from left to right and perform two kinds of actions: shifting a symbol onto the stack, and reduction of the stack without reading any symbol.

GLR parsers operate on the same principles as LR parsers, but may be used with all CFGs, including ambiguous ones. A LR parser may reach a point where it cannot decide what action to take; a GLR parser forges ahead, simulating nondeterminism and effectively taking all actions.

Why forsake an efficient deterministic algorithm for GLR parsing? Functionality. Grammars for common programming languages, like C++, are ambiguous and thus cannot be handled directly by deterministic algorithms. Graham/Glanville code generation [9] employs highly-ambiguous grammars, and so may software reengineering, such as a grammar which can recognize several dialects of a programming language [6]. Some of the above grammars may be

shoehorned into a form usable by deterministic parsing algorithms. Doing so, however, has some undesirable side effects: development time is increased, another source of potential bugs is introduced, maintenance is made more difficult, and programmers are required to have expertise in the minutiæ of parsing algorithms. Using general parsing algorithms such as GLR parsing avoids these problems entirely.

Hidden left recursion [18–20] in grammars is known to present a problem for GLR parsers. This is because a grammar with hidden left recursion has an infinite number of ways to derive certain strings in the language it generates. The same difficulty has been noted when modifying LR parsers to handle ambiguous grammars [23]. Hidden left recursion may be detected at parser-generation time.

Right recursion in a grammar may similarly be detected at parser-generation time, and may be mechanically removed. Given a grammar without right and hidden left recursions, we prove in this paper that the number of consecutive reductions between the shifts of two adjacent symbols cannot be greater than a constant in a GLR parser.

This GLR "boundedness" property can be applied to further speed up our GLR parser [4, 3] by ensuring that every transition of the parsing automaton consumes one input symbol. Moreover, the number of automaton states is reduced. Our timings show that, especially for highly ambiguous grammars, the parser is significantly faster than a standard GLR parser.[1]

2 Definitions

2.1 Languages, Grammars, Recursion

We use standard notation as defined in texts such as [2], [10], or [21].

Let an *alphabet* be a finite set of symbols. A *language* over an alphabet T is a set of strings over T. The notation T^* denotes the set of all strings over T including the empty string, denoted by ε. Set T^+ is defined as $T^+ = T^* \setminus \{\varepsilon\}$. Similarly for string $\alpha \in T^*$, the notation α^m, $m \geq 0$, denotes the m-fold concatenation of α with $\alpha^0 = \varepsilon$. Set α^* is defined as $\alpha^* = \{\alpha^m : m \geq 0\}$ and $\alpha^+ = \alpha^* \setminus \{\varepsilon\}$.

A *context-free grammar* (CFG) is a 4-tuple $G = (N, T, P, S)$, where N and T are finite disjoint sets of *nonterminal* and *terminal symbols*, respectively. P is a finite set of *rules* $A \to \alpha$, where $A \in N$, $\alpha \in (N \cup T)^*$. $S \in N$ is the *start symbol*. The *augmented grammar* G' derived from a CFG G is defined as $G' = (N', T', P', S')$, where $N' = N \cup \{S'\}$, $P' = P \cup \{S' \to \vdash S \dashv\}$, and $T' = T \cup \{\vdash, \dashv\}$. In most treatments of LR parsing, an augmented grammar has a new starting rule of the form $S' \to S$. To ensure that the first and the last parsing operation are shifts, we have modified this starting rule to $S' \to \vdash S \dashv$. Thus, input strings are enclosed in markers $\vdash$ and $\dashv$ which delimit their left and right ends, respectively.

[1] An early presentation of this idea can be found in [11].

Relation $\Rightarrow$ is called *derivation*: if $\alpha A\gamma \Rightarrow \alpha\beta\gamma$, $A \in N$, and α, β, $\gamma \in (N \cup T)^*$, then rule $A \to \beta$ is in P. Symbols $\Rightarrow^+$, and $\Rightarrow^*$ are used for the *transitive*, and the *reflexive transitive* closure of $\Rightarrow$, respectively. A *rightmost derivation* $\Rightarrow_{\mathrm{rm}}$ is a relation $\alpha A\,\gamma \Rightarrow \alpha\beta\gamma$, where $\gamma \in T^*$. *Recursion* is a relation $A \Rightarrow^+ \alpha A\beta$. *Right recursion* is a relation $A \Rightarrow^+ \alpha A$. *Left recursion* is a relation $A \Rightarrow^+ A\beta$. *Hidden-left recursion* is a relation $A \Rightarrow^+ B\alpha A\beta$, where $B\alpha \Rightarrow^+ \varepsilon$. α is a *sentential form* if $S \Rightarrow^* \alpha$.

The language generated by a CFG G, denoted by $L(G)$, is the set of strings $L(G) = \{w : S \Rightarrow^* w,\, w \in T^*\}$. A *right parse* of a string w is the sequence of rules used in a derivation $S \Rightarrow^+_{\mathrm{rm}} w$, in reverse order.

A *derivation tree* is a labelled, ordered tree representing a syntactic structure of a string w generated by the grammar G. Its root is labelled by the start symbol S and its leaves are labelled by empty strings or terminal symbols. Each interior node of the tree is labelled by a nonterminal symbol A, and the children of such a node are labelled, from left to right, by symbols from the right-hand side β of a rule $A \to \beta \in P$. A derivation $S \Rightarrow^* w$ corresponds to a derivation tree whose leaves, if read from left to right, are labelled with the string w.

Finally, the *first set* $\mathrm{FIRST}_k(\alpha) = \{x \in T^* : \alpha \Rightarrow^* x\beta$ and $|x| = k$, or $\alpha \Rightarrow^* x$ and $|x| < k\}$, where $k \geq 0$.

2.2 GLR Parsing

Given a string $\vdash w \dashv$, a *GLR parser* for a CFG $G = (N, T, P, S)$, where $G' = (N', T', P', S')$ is the augmented grammar of G, reads the string $\vdash w \dashv$ from left to right without any backtracking and produces all right parses of w in G.

The parsing algorithm can be thought of as a non-deterministic pushdown automaton. For efficiency, Tomita's algorithm [25] represents multiple stacks using a *graph-structured stack*, a directed acyclic graph which allows common stack contents to be shared.

A configuration of the parser at a given instant is a triple (ζ, x, π), where ζ is a string of stack symbols, x is the unread part of the string $\vdash w \dashv$, and π is the created part of a right parse of w.

A string γ is a *viable prefix* of G if γ is a prefix of $\alpha\beta$, and $S \Rightarrow^*_{\mathrm{rm}} \alpha Ax \Rightarrow_{\mathrm{rm}} \alpha\beta x$ is a rightmost derivation in G; the string β is called the *handle* of the last derivation step. We use the term *viable string* to refer to $\alpha\beta$ in its entirety. During parsing, every possible stack configuration ζ corresponds to a viable prefix, which will be denoted by $\mathrm{cvp}(\zeta)$ (cvp stands for *c*orresponding *v*iable *p*refix).

Relation $\models$ denotes a *move* between two configurations. The parser performs two kinds of moves:

1. When the stack contents correspond to a viable prefix containing an incomplete handle, the parser performs a *shift* $(\zeta, ay, \pi) \models (\zeta a', y, \pi)$. The shift reads one symbol a and pushes a symbol corresponding to a onto the stack. $\mathrm{cvp}(\zeta a') = \mathrm{cvp}(\zeta)a$.
2. When the stack contents correspond to a viable prefix containing the handle β, the parser performs a *reduction* $(\zeta, ay, \pi) \models (\eta A', ay, \pi j)$ by the j-th

rule $A \to \beta$. The reduction pops $|\beta|$ symbols from the top of the stack and pushes a symbol corresponding to A onto the stack. $\text{cvp}(\eta A') = \alpha A$, where $\text{cvp}(\zeta) = \alpha\beta$.

Several different moves may be possible from a given configuration. The stack contents may hold multiple viable strings, signalling the need for multiple reductions (a *reduce/reduce conflict*). Or, the stack may contain viable prefixes with both complete and incomplete handles, necessitating both kinds of move (a *shift/reduce conflict*).

The stack symbols refer to state numbers in a parsing automaton for G; consequently, there can be many stack symbols corresponding to a symbol of the grammar. The symbol to be pushed onto the stack is determined according to the parsing table, which the parser is driven by.

The initial configuration is the triple $(\#, \vdash w \dashv, \varepsilon)$, where $\#$ is the initial stack symbol and $\text{cvp}(\#) = \varepsilon$. The final, accepting configuration is $(\# \vdash' Z \dashv', \varepsilon, \pi)$, where $(\#, \vdash w \dashv, \varepsilon) \models^+ (\# \vdash' Z \dashv', \varepsilon, \pi)$, $\text{cvp}(\# \vdash' Z \dashv') = \vdash S \dashv$, and π is a right parse of w.

[2], [20], [25], and [26] provide further information on GLR parsing.

3 Reductions between Shifts of Two Adjacent Symbols

Grammars without right and hidden left recursions have a property which proves to be useful in the context of GLR parsing. Namely, after shifting an input symbol, the number of reductions that can take place before shifting the next input symbol is bounded. In this section we prove this property, as well as a companion theorem which shows that this useful property does not hold if either of the grammar restrictions is violated.

Theorem 1. *Given a generalized LR parser for a CFG $G = (N, T, P, S)$ without right and hidden left recursions, the number of consecutive reductions between the shifts of two adjacent symbols cannot be greater than a constant.*

Proof. Assume an m-th move is a reduction $(\zeta_m, y, \pi) \models (\eta_m {A_m}', y, \pi j)$ by the j-th rule $A_m \to \beta_m$, which means that $S' \Rightarrow^*_{\text{rm}} \alpha_m A_m y \Rightarrow_{\text{rm}} \alpha_m \beta_m y$, where $\alpha_m \beta_m = \text{cvp}(\zeta_m)$, $\alpha_m A_m = \text{cvp}(\eta_m A'_m)$. The next, $m+1$-th, move is again a reduction by a rule $A_{m+1} \to \beta_{m+1}$ iff $S' \Rightarrow^*_{\text{rm}} \alpha_{m+1} A_{m+1} y \Rightarrow_{\text{rm}} \alpha_{m+1} \beta_{m+1} y$, where $\alpha_{m+1}\beta_{m+1} = \alpha_m A_m$.

This means that k, $k \geq 2$, consecutive reductions, the first of which is an i-th move, between the shifts of two adjacent symbols implies the derivation

$$S' \Rightarrow^*_{\text{rm}} \alpha_{i+k-1} A_{i+k-1} y \tag{1}$$

followed by the derivation

$$\begin{aligned} \alpha_{i+k-1} A_{i+k-1} y &\Rightarrow_{\text{rm}} \alpha_{i+k-1} \beta_{i+k-1} y = \alpha_{i+k-2} A_{i+k-2} y \\ &\Rightarrow_{\text{rm}} \cdots \end{aligned} \tag{2}$$

$$\Rightarrow_{\rm rm}\ \alpha_{i+2}\beta_{i+2}y = \alpha_{i+1}A_{i+1}y$$
$$\Rightarrow_{\rm rm}\ \alpha_{i+1}\beta_{i+1}y = \alpha_{i}A_{i}y$$
$$\Rightarrow_{\rm rm}\ \alpha_{i}\beta_{i}y$$

Thus, we must prove that k is bounded by a constant. In Derivation 2, every sentential form $\alpha_l A_l y$, $i \leq l \leq i+k-1$, has the same suffix $y \in T^*$ following the nonterminal A_l. Since the derivation is the rightmost, the only possible recursion in Derivation 2 would have been a right one, which contradicts our assumption about the grammar.

Let k_B, where $B = A_l$, $i \leq l \leq i+k-1$, denote the number of derivation steps in Derivation 2 that generate the derivation subtree with the root B; formally, k_B is the greatest integer such that $\alpha_{l-p}\beta_{l-p} = \alpha_l \gamma_{l-p}$ for each $p \in \langle 0, k_B - 1\rangle$. Let Γ_B denote the string γ_{l-k_B-1}, which is generated by B. An important property is that k_B can never be greater than a given constant because Derivation 2 contains no recursion as is shown above.

Let $B_1 = A_{i+k-1}$. One of the possibilities is that $k = k_{B_1}$. Otherwise, if k is greater than k_{B_1}, then $\Gamma_{B_1} = \varepsilon$, and $\alpha_{i+k-1}A_{i+k-1} = \alpha_{i+k-1-k_{A_{i+k-1}}}B_2B_1$, where $B_2 = A_{i+k-1-k_{B_1}}$. Informally, one of the possibilities is that Derivation 2 generates only a derivation subtree with the root B_1. Otherwise, B_1 generates the empty string in k_{B_1} derivation steps and $\alpha_{i+k-1}A_{i+k-1}$ ends with B_2B_1. Then, supposing $\Gamma_{B_1} = \varepsilon$, there is again the possibility that $k = k_{B_1} + k_{B_2}$. Otherwise, if k is greater than $k_{B_1} + k_{B_2}$, then $\Gamma_{B_2} = \varepsilon$, and $\alpha_{i+k-1}A_{i+k-1} = \alpha_{i+k-1-k_{B_1}-k_{B_2}}B_3B_2B_1$, where $B_3 = A_{i+k-1-k_{B_1}-k_{B_2}}$.

In general, if k is greater than $k_{B_1} + k_{B_2} \ldots + k_{B_{n-1}}$, where $n \geq 2$,

$$\alpha_{i+k-1}A_{i+k-1} = \alpha_{i+k-1-k_{B_1}-k_{B_2}-\ldots-k_{B_{n-1}}}B_n \ldots B_2B_1,$$

$$\begin{array}{rclcrcl}
B_1 & = & A_{i+k-1}, & & \Gamma_{B_1} & = & \varepsilon, \\
B_2 & = & A_{i+k-1-k_{B_1}}, & & \Gamma_{B_2} & = & \varepsilon, \\
B_3 & = & A_{i+k-1-k_{B_1}-k_{B_2}}, & & & \ldots & \\
 & \ldots & & & \Gamma_{B_{n-1}} & = & \varepsilon, \\
B_n & = & A_{i+k-1-k_{B_1}-k_{B_2}-k_{B_{n-1}}} & & & &
\end{array}$$

then there is the possibility that $k = k_{B_1} + k_{B_2} \ldots + k_{B_n}$. Otherwise, if k is greater than $k_{B_1} + k_{B_2} \ldots + k_{B_n}$, then $\Gamma_{B_n} = \varepsilon$, and

$$\alpha_{i+k-1}A_{i+k-1} = \alpha_{i+k-1-k_{B_1}-k_{B_2}-\ldots-k_{B_n}}B_{n+1}B_n \ldots B_2B_1,$$

where $B_{n+1} = A_{i+k-1-k_{B_1}-k_{B_2}-k_{B_n}}$. We recall that each of $k_{B_1}, k_{B_2}, \ldots, k_{B_n}$ is bounded by a given constant. It means that k could be unbounded by a constant only if n is an unbounded integer, which contradicts the following property:

If $\alpha_{i+k-1}A_{i+k-1} = \alpha_{i+k-1-k_{B_1}-k_{B_2}-\ldots-k_{B_{n-1}}}B_n \ldots B_2B_1$, where $n \geq 2$, and $B_1, B_2, \ldots, B_n \Rightarrow^+ \varepsilon$, then n cannot be greater than a given constant because no hidden left recursion is possible in Derivation 1.

Thus, k is bounded by a constant. □

Theorem 2. *Given a generalized LR parser for a CFG $G = (N, T, P, S)$ with a right or a hidden left recursion, the number of consecutive reductions between the shifts of two adjacent symbols is not bounded by a constant.*

Proof. The property to be proved is well known for the case of a grammar with a hidden left recursion $A \Rightarrow^+ B\alpha A\beta$, where $B\alpha \Rightarrow^+ \varepsilon$, because it is not known how many times $B\alpha \Rightarrow^+ \varepsilon$ should be reduced before the shift of the first symbol from the subtree with the root A [18–20].

For the case of right recursion we will assume a right recursion

$$A_0 \Rightarrow_{\text{rm}} \beta_1 A_1 \Rightarrow_{\text{rm}} \beta_1\beta_2 A_2 \Rightarrow_{\text{rm}} \ldots \Rightarrow_{\text{rm}} \beta_1\beta_2\ldots\beta_k A_k$$

where $S' \Rightarrow^*_{\text{rm}} \alpha A_k y$, $k \geq 1$, $A_k = A_0$, $A_m \neq A_0$, $1 \leq m < k$, and $A_{i-1} \to \beta_i A_i$ is the j_i-th rule in P, $1 \leq i < k$. The recursion implies the derivations

$$S' \Rightarrow^*_{\text{rm}} \alpha A_k y \Rightarrow^+_{\text{rm}} \alpha\beta A_k y \Rightarrow^+_{\text{rm}} \alpha\beta^2 A_k y \Rightarrow^+_{\text{rm}} \ldots \Rightarrow_{\text{rm}} \alpha\beta^l A_k y$$

where $l \geq 1$, and $\beta = \beta_1\beta_2\ldots\beta_k$.

This derivation means that there is a sequence of consecutive reductions

$$\begin{aligned}
(\zeta(\eta_1\eta_2\ldots\eta_k)^l A'_k, y, \mu) &\models (\zeta(\eta_1\eta_2\ldots\eta_k)^{l-1}\eta_1\eta_2\ldots\eta_{k-1}A'_{k-1}, y, \mu j_k) \\
&\models \ldots \\
&\models (\zeta(\eta_1\eta_2\ldots\eta_k)^{l-1}\eta_1 A'_1, y, \mu j_k\ldots j_2) \\
&\models (\zeta(\eta_1\eta_2\ldots\eta_k)^{l-1} A'_k, y, \mu j_k\ldots j_2 j_1) \\
&\models \ldots \\
&\models (\zeta(\eta_1\eta_2\ldots\eta_k) A'_k, y, \mu(j_k\ldots j_2 j_1)^{l-1}) \\
&\models \ldots \\
&\models (\zeta(\eta_1) A'_1, y, \mu(j_k\ldots j_2 j_1)^{l-1} j_k\ldots j_2) \\
&\models (\zeta A'_k, y, \mu(j_k\ldots j_2 j_1)^l)
\end{aligned}$$

where $\text{cvp}(\zeta) = \alpha$, $\text{cvp}(\eta_n) = \beta_n$, $\text{cvp}(A'_n) = A_n$, $1 \leq n \leq k$.

Since l is unbounded, the theorem holds. □

We note a related result for "strong" LR parsing, deterministic parsing for "strong LR" grammars [16]. A grammar is said to be *strong LR*(k) if the three conditions:

1. $S' \Rightarrow^*_{\text{rm}} \alpha_1 A_1 w_1 \Rightarrow_{\text{rm}} \alpha_1\beta_1 w_1 = \alpha'_1 X w_1$
2. $S' \Rightarrow^*_{\text{rm}} \alpha_2 A_2 w_2 \Rightarrow_{\text{rm}} \alpha_2\beta_2 x = \alpha'_2 X w_2$
3. $\text{FIRST}_k(w_1) = \text{FIRST}_k(w_2)$

imply that $A_1 = A_2, \beta_1 = \beta_2$, and $x = w_2$. In [17, 22] it was observed that, for strong LR parsing, the number of reductions that can take place between shifts of two adjacent input symbols is bounded.

4 Faster GLR Parsing

4.1 Towards a Faster GLR Parser

GLR parsers are tied heavily to operations on the graph-structured stack. In fact, a dominant factor in the time complexity of the GLR algorithm is searching for all paths of a certain length from a stack node [12]. We reasoned that if the number of stack operations were to be reduced, then a faster GLR parser would result. Since LR parsing is the basis of GLR parsing, we began there.

Let Φ represent the set of all viable strings for a given CFG. If we construct a trie [14] from the elements of Φ, then we have a data structure that can be used in the same manner as the finite automaton in a traditional LR parser: we can step through the trie as input symbols are consumed and, upon reaching a trie leaf, know that we have seen a handle. This is the same information that we would get from the traditional LR parser's automaton. However, a trie has the additional property that a path from root to leaf is unique. This means that we not only know what handle we been found, but the complete viable string we have found. In contrast, a traditional LR parser would use the stack to retain this information. We are thus able to reduce reliance on the stack.

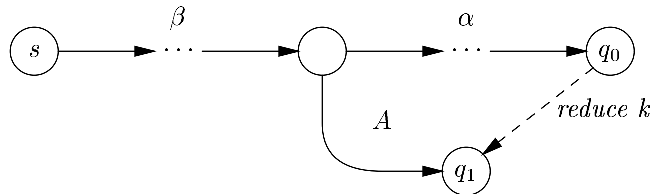
Our trie has one slight problem, however: it can be infinite in size! The set of viable prefixes, and consequently Φ, is known to be a regular set [13]. Unfortunately, this set will be infinite in all but the most trivial cases.

If we had no recursion in the grammar G, then $L(G)$ would be finite and so would Φ. We induce this situation by introducing *limit points* at nonterminal symbols in G in such a way as to break the recursive cycles in G. The contents of Φ can be generated by a new grammar, F, which can be derived automatically from G [5]. F is a regular grammar by construction [5], and as such is equivalent to a finite automaton [10]. Removing a set of transitions from this finite automaton so that it no longer contains cycles makes it accept a finite language [10], and yields a set of limit points.

Removing a set of transitions from a finite automaton can be modelled by the feedback arc set problem [24]. Ideally, we would like to find the *minimum* number of limit points for a grammar. Finding the minimum feedback arc set is NP-hard [8], but heuristic approximation algorithms such as [7] exist.

Not all recursive cycles in G must be broken. Left recursion requires no such special treatment, because left recursive grammar rules do not yield an unbounded number of elements in Φ.

Once a finite trie is constructed, it can be converted into a finite automaton by adding "reduction transitions," which indicate reduction by a particular grammar rule. We 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, we 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 $\vdash S \dashv$. This finite automaton accepts a subset of $L(G)$ due to the limit points; it is insufficiently powerful to recognize all of $L(G)$, since it cannot recognize and remember enough information.

The places where the finite automaton needs to remember additional information have already been identified: the limit points. At these places in our finite automaton, we push additional information onto the stack. This gives us a pushdown automaton, which is powerful enough to recognize $L(G)$ as described by the CFG G.

Having pushed information onto the stack, our automaton jumps to a automaton "subroutine" which recognizes a subset of G's rules. If a limit point had been set at the nonterminal A, for instance, then there would be a subroutine to recognize G_A.

For example, consider a CFG for simple arithmetic expressions. The limit point is indicated by the boxed nonterminal:

$$
\begin{aligned}
&(0)\ S' \rightarrow \vdash E \dashv \\
&(1)\ E \rightarrow E + F \\
&(2)\ E \rightarrow F \\
&(3)\ F \rightarrow a \\
&(4)\ F \rightarrow (\ \boxed{\mathrm{E}}\ )
\end{aligned}
$$

Our automaton for this grammar is shown in Fig. 1. Figure 2 shows the sequences of moves performed by a standard LR parser and our automaton for the string $\vdash a+((a+a)) \dashv$. As claimed, the number of stack operations is greatly reduced in comparison to the standard LR parser.

Use of our LR automaton in a GLR parser is straightforward. As mentioned, a standard GLR parser starts a new parsing process when the underlying LR parser reaches a point where it must perform conflicting actions. In a standard LR parser, two types of conflict are possible: shift/reduce and reduce/reduce. The principle is exactly the same when using our LR automaton in a GLR parser, except there are two additional types of action – stack pushes and pops. All combinations except for shift/shift and pop/pop conflict with each other. While these extra sources of conflict may seem to create more work for the GLR parser, recall that stack actions in our automata are designed to be infrequent; in practice, the extra conflicts do not present a problem.

4.2 Optimizing Our GLR Parser

Our parser described in the previous section performs shift, reduce, push, and pop actions between its states. In the parsing automaton, actions are represented

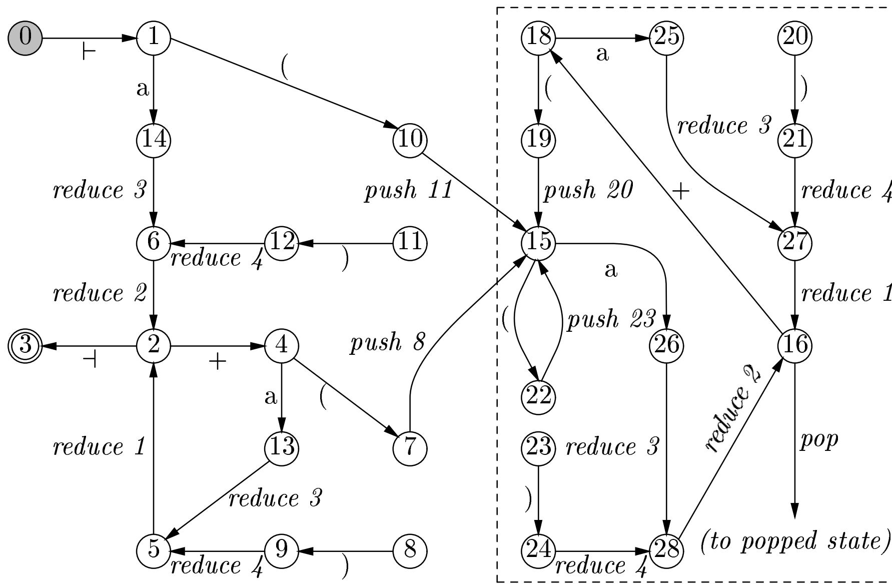


Fig. 1. Our pushdown automaton for the expression grammar. The subroutine recognizing G_E is enclosed in the dashed box, and the grayed circle is the start state.

Stack	Input	Output
$\#$	$\vdash a + ((a + a)) \dashv$	ε
$\# \vdash$	$a + ((a + a)) \dashv$	ε
$\# \vdash a$	$+ ((a + a)) \dashv$	ε
$\# \vdash F_1$	$+ ((a + a)) \dashv$	3
$\# \vdash E_1$	$+ ((a + a)) \dashv$	32
$\# \vdash E_1 +$	$((a + a)) \dashv$	32
$\# \vdash E_1 + ($	$(a + a)) \dashv$	32
$\# \vdash E_1 + (($	$a + a)) \dashv$	32
$\# \vdash E_1 + ((a$	$+ a)) \dashv$	32
$\# \vdash E_1 + ((F_1$	$+ a)) \dashv$	323
$\# \vdash E_1 + ((E_2$	$+ a)) \dashv$	3232
$\# \vdash E_1 + ((E_2 +$	$a)) \dashv$	3232
$\# \vdash E_1 + ((E_2 + a$	$)) \dashv$	3232
$\# \vdash E_1 + ((E_2 + F_2$	$)) \dashv$	32323
$\# \vdash E_1 + ((E_2$	$)) \dashv$	323231
$\# \vdash E_1 + ((E_2)$	$) \dashv$	323231
$\# \vdash E_1 + (F_2$	$) \dashv$	3232314
$\# \vdash E_1 + (E_2$	$) \dashv$	32323142
$\# \vdash E_1 + (E_2)$	$\dashv$	32323142
$\# \vdash E_1 + F_2$	$\dashv$	323231424
$\# \vdash E_1$	$\dashv$	3232314241
$\# \vdash E_1 \dashv$	ε	3232314241

Stack	State	Input	Output
	0	$\vdash a + ((a + a)) \dashv$	ε
	1	$a + ((a + a)) \dashv$	ε
	14	$+ ((a + a)) \dashv$	ε
	6	$+ ((a + a)) \dashv$	3
	2	$+ ((a + a)) \dashv$	32
	4	$((a + a)) \dashv$	32
	7	$(a + a)) \dashv$	32
8	15	$(a + a)) \dashv$	32
8	22	$a + a)) \dashv$	32
8 23	15	$a + a)) \dashv$	32
8 23	26	$+ a)) \dashv$	32
8 23	28	$+ a)) \dashv$	323
8 23	16	$+ a)) \dashv$	3232
8 23	18	$a)) \dashv$	3232
8 23	25	$)) \dashv$	3232
8 23	27	$)) \dashv$	32323
8 23	16	$)) \dashv$	323231
8	23	$)) \dashv$	323231
8	24	$) \dashv$	323231
8	28	$) \dashv$	3232314
8	16	$) \dashv$	32323142
	8	$) \dashv$	32323142
	9	$\dashv$	32323142
	5	$\dashv$	323231424
	2	$\dashv$	3232314241

Fig. 2. Trace of standard LR parser (left), and our automaton (right).

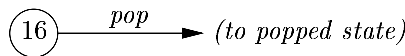
by edges and states are represented by nodes. We will demonstrate that every cycle in the automaton must contain at least one shift edge if the grammar is without right and hidden left recursions. Consequently, the automaton can be optimized so that, on every move, it consumes one input symbol, changes the top of the stack, and emits an output. Compared to our unoptimized automaton, the optimized one has fewer states and performs fewer moves, although there is potentially more work per transition.

The key to automaton optimization is this: in the automaton, no cycle consisting only of reduce, push and pop edges can exist. The push and pop actions jump to and return from a subroutine representing a recursive nonterminal in the place of a limit point which breaks a recursion. As described in Sect. 4.1, that recursion is never left recursion, and therefore at least one shift must be performed between two jumps to the same subroutine. Every cycle containing a push edge must therefore also contain a shift edge.

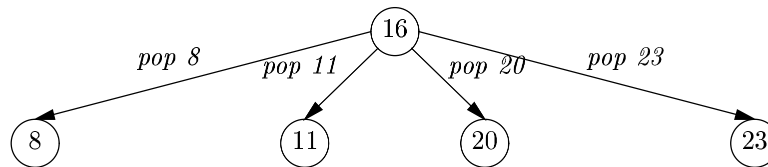
Can a cycle exist consisting only of reduce and pop edges? The pop action always follows the reduce action, which means that cycles consisting only of pop edges cannot exist. Cycles consisting only of reduce edges cannot exist because the number of consecutive reductions is bounded by a constant as shown in Theorem 1. For the same reason no cycles containing one or more reduce and one or more pop edges can exist in the parser — the number of the symbols to be popped from the stack can generally be unbounded because every subroutine can jump to itself recursively. Thus, a cycle consisting only of reduce, push and pop edges cannot exist.

Given this, we optimize our GLR parsing automaton in the following manner, using Fig. 1 as a running example. For clarity, we refer to our unoptimized automaton as P, and the optimized one as P'.

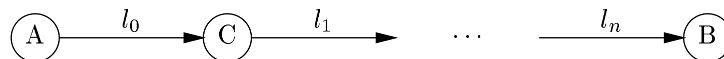
1. We replace all the *pop* edges in P with edges *pop* X that lead directly to all possible destination states X. During parsing, a *pop* X edge is only traversed if the topmost state on the stack is X. In our example, we replace edge



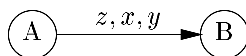
   with edges



2. We create P' from P's final state and the states of P from which shift and pop are performed. In our example, we create P' from states 0, 1, 2, 3, 4, 8, 11, 15, 16, 18, 20 and 23. Then, for every sequence of edges

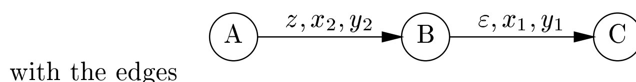


in P, where $n \geq 0$, l_0 is shift or pop, $l_1, \ldots, l_n$ are reduce or push (there may be more than one push), and A and B are states of P', we create an edge in P'

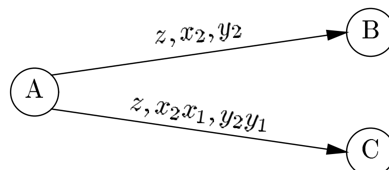


where z is the input to be read, x is the change of the stack contents and y is the output emitted by the actions $l_0, l_1, \ldots, l_n$.

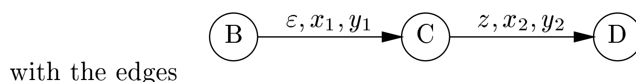
3. We join edges in P' so that every edge reads one symbol. While P' contains a state B with outgoing edges that consume no symbols, we apply one of the two transformations below.
   (a) Replace every sequence of edges



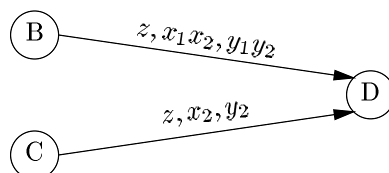
with the edges



   (b) Replace every sequence of edges



with the edges



   Upon completion of this step, every edge of P' will read one symbol. We choose between Transformations (a) and (b) to try and maximize the number of useless states in P'. A state is *useless* if it is unreachable from the start state, or if the final state cannot be reached from it. For example, using Transformation (b) at state 16 makes states 8, 11, 20, and 23 useless.
4. We remove useless states and their corresponding edges from P'.

The resulting optimized automaton is shown in Fig. 3. We observe that the change x of the stack is always of the form $-x_1 \; +x_2$, where x_1 and x_2 are sequences (possibly empty) of popped and pushed stack symbols, respectively, because all the other forms can easily be precomputed to this form.

Since every cycle in P must contain at least one shift edge, the number of edges between any two states in the resulting P' must be finite. The newly constructed edges merely simulate actions performed by P, and so the optimized parser is correct.

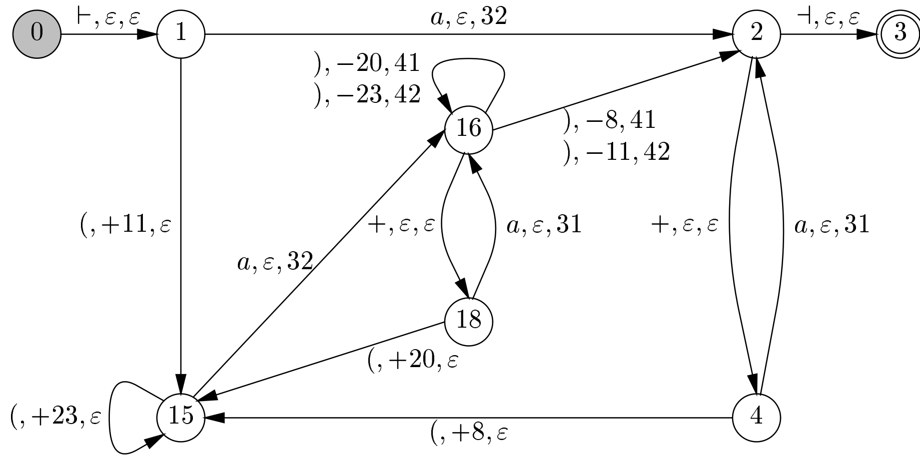


Fig. 3. The optimized pushdown automaton. Each edge is labelled by a triple a, x, y, where a is the symbol to be read, x is the change of the stack, and y is the output.

Given a string $\vdash a + ((a + a)) \dashv$, its parsing by the optimized parser from Fig. 3 performs the sequence of moves shown in Fig. 4. This is a 47% reduction in state transitions from the standard LR parser, and a 54% reduction in transitions from our original automaton.

Stack	State	Input	Output
	0	$\vdash a + ((a + a)) \dashv$	ε
	1	$a + ((a + a)) \dashv$	ε
	2	$+ ((a + a)) \dashv$	32
	4	$((a + a)) \dashv$	32
8	15	$(a + a)) \dashv$	32
8 23	15	$a + a)) \dashv$	32
8 23	16	$+ a)) \dashv$	3232
8 23	18	$a)) \dashv$	3232
8 23	16	$)) \dashv$	323231
8	16	$) \dashv$	32323142
	2	$) \dashv$	3232314241
	3	ε	3232314241

Fig. 4. Trace of optimized pushdown automaton.

5 Some Empirical Results

We compared implementations of a standard GLR parser and our unoptimized GLR automata with automata optimized as described in the previous section. All implementations were written in Python, an object-oriented scripting language; all timings were conducted on a 200 MHz Pentium with 64 M of RAM running Debian GNU/Linux version 2.1 and Python 1.5.2. As Fig. 5 shows, our optimized automata are substantially faster for these highly ambiguous grammars.

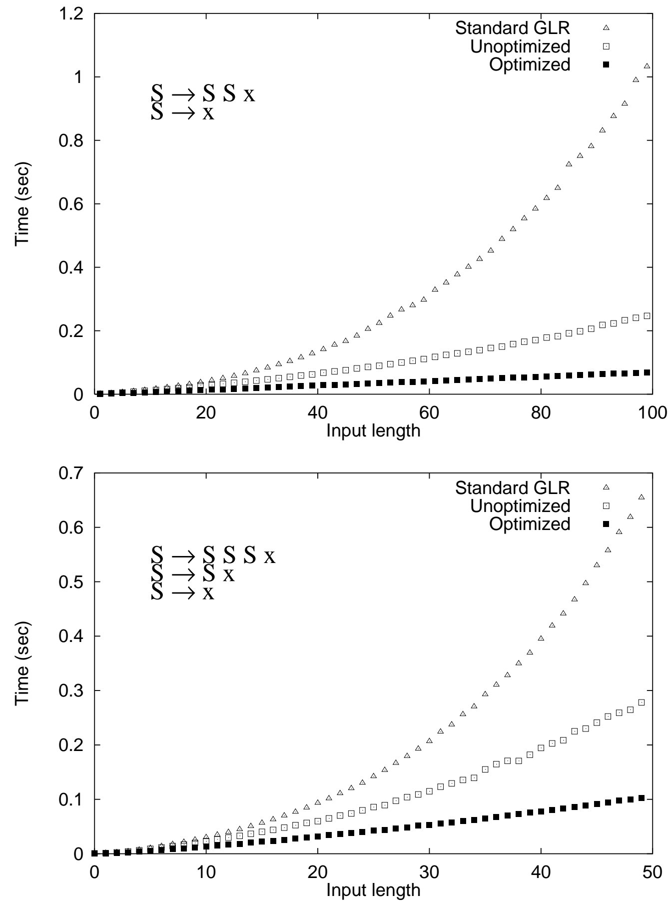


Fig. 5. Timing results for some ambiguous grammars.

In practice, grammars tend to be mostly unambiguous, with slight local ambiguities. This idea has been noted – and exploited – in work by Lang [15], Tomita [25, 26], and others [27]. The performance of our optimized automata on

unambiguous grammars is thus of interest too. Figure 6 demonstrates that our optimized automaton easily outperforms the other implementations. The construction of a parser generator is underway, which will allow us to conduct more extensive timing experiments.

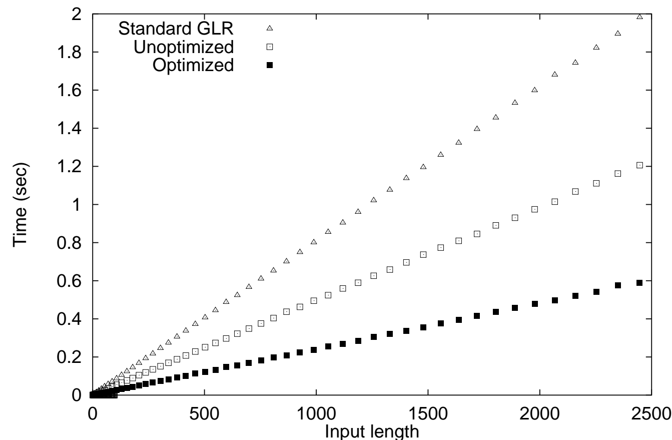


Fig. 6. Timing results for the expression grammar.

6 Reconstructing Derivations

While the focus of our work has been improving recognition performance, it would be futile if all the derivations of the input could not be reconstructed. In this section we discuss how this may be accomplished.

For an unambiguous LR grammar, the situation is simple: it is sufficient to know the ordered sequence of reductions made by the parser. This is a special case, however. There is an implicit piece of information tying these reductions together, namely the knowledge that there can only be a single derivation of the input. In other words, all the reductions must belong to the same derivation.

This leads to the general case. Recall that a GLR parser conceptually implements multiple LR parsers running in parallel, each producing a derivation of the input. If every conceptual LR parser were simply to indiscriminately report reductions, then the result would be a tangled mess; we have to have some means of associating reductions together that belong to the same derivation.

To this end, we create a *reduction log*, a directed acyclic graph to track sequences of reductions. (This is distinct from the graph-structured stack used for recognition.) Each GLR parsing process has a pointer to some node in the reduction log, indicating the last reduction performed by that process. Nodes in the reduction log may be one of two types:

1. Reduction nodes. There is one reduction node for every reduction reported by the parser. A reduction node contains two pieces of information: a number indicating the grammar rule that was reduced by, and a pointer to a previous node in the reduction log.
2. Fan-in nodes. When two parser processes merge in the GLR parser by virtue of being in the same automaton state, we note this event in the reduction log with a fan-in node. A fan-in node does not correspond to a parser reduction, and only contains pointers to previous nodes in the reduction log.

Following recognition, it is a simple matter to negotiate the reduction log and extract some or all derivations of the input. Figure 7 shows a reduction log and the derivations it represents.

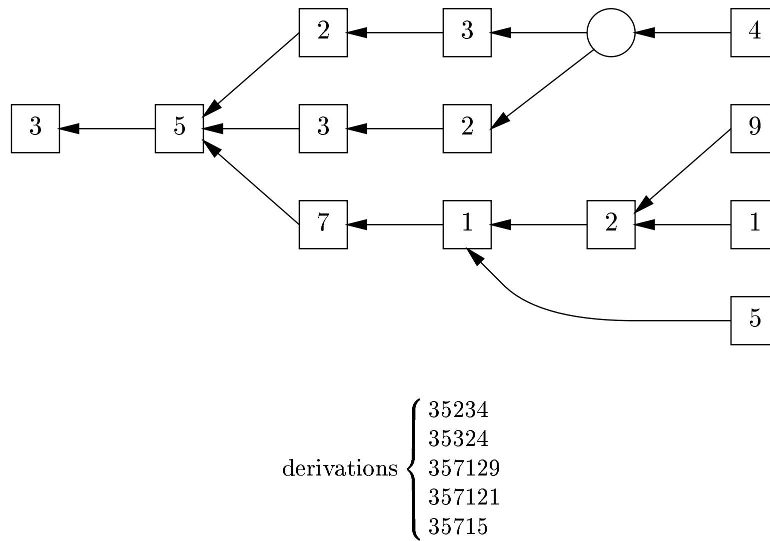


Fig. 7. A sample reduction log and derivations. Squares are reduction nodes and circles are fan-in nodes.

7 Concluding Remarks

We have proven a property of generalized LR parsing and demonstrated how this property may be used to construct a faster GLR parser, which reads one symbol on every transition and performs significantly less stack operations, for grammars without right and hidden left recursions. The timings show a dramatic speed improvement compared to other GLR implementations.

References

1. Aho, A.V., Sethi, R., Ullman, J.D. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass., 1986.
2. Aho, A.V., Ullman, J.D. *The Theory of Parsing, Translation and Compiling.* Vol. 1: Parsing, Vol. 2: Compiling, Prentice-Hall, New York, 1972.
3. Aycock, J.D., Horspool, R.N. *Faster Generalized LR Parsing.* In: LNCS 1575 (Compiler Construction), Springer-Verlag, Berlin, Heidelberg, 1999.
4. Aycock, J.D. *Faster Tomita Parsing.* MSc Thesis, University of Victoria, 1998.
5. Backhouse, R. C. An Alternative Approach to the Improvement of *LR(k)* Parsers. *Acta Informatica*, 6:277–296, 1976.
6. van den Brand, M., Sellink, A., and Verhoef, C. Current Parsing Techniques in Software Renovation Considered Harmful. *International Workshop on Program Comprehension*, 1998, pp. 108–117.
7. Eades, P., Lin, X., and Smyth, W. F. A fast and effective technique for the feedback arc set problem. *Information Processing Letters*, 47:319–323, 1993.
8. Garey, M. R. and Johnson, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.
9. Glanville, R. S., and Graham, S. L. A New Method for Compiler Code Generation. *Fifth Annual ACM Symposium on Principles of Programming Languages*, 1978, pp. 231–240.
10. Hopcroft, J.E., Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Mass., 1979.
11. Janoušek, J. *The Very Fast Generalized LR Parsing for CFGs without Right Recursion.* In: Int. Student's Conference POSTER '99, CTU Press, Prague, pp. IC 12, 1999.
12. Kipps, J. R. GLR Parsing in Time $O(n^3)$. In Tomita [26], pp. 43–59.
13. Knuth, D. E. On the Translation of Languages from Left to Right. *Information and Control*, 8:607–639, 1965.
14. Knuth, D. E. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, 1973.
15. Lang, B. Deterministic Techniques for Efficient Non-Deterministic Parsers. In *Automata, Languages, and Programming (LNCS #14)*, J. Loeckx, ed., Springer-Verlag, 1974, pp. 255–269.
16. Melichar, B. Strong LR Grammars. In: *Compiler Compiler and High Speed Compilation*, Proceedings of the Workshop, Berlin, pp. 10–14, 1988.
17. Melichar, B., Šaloun, P. *New Approaches to Sequential Strong LR(1) Parsing.* In: Workshop '97, CTU Press, Prague, pp. 277–278, 1997.
18. Nederhof, M.J., Sarbo, J.J. *Increasing the Applicability of LR Parsing.* In: 3rd Int. Workshop on Parsing Technologies IWPT '93, Tilburg and Durbuy, pp. 187–201, 1993.
19. Nozohoor-Farshi, R. *Handling of Ill-designed Grammars in Tomita's Parsing Algorithm.* In: Int. Workshop on Parsing Technologies IWPT '89, Carnegie Mellon University, Pittsburgh, pp. 182–192, 1989.
20. Sikkel, K. *Generalized LR Parsing.* Chapter 12 in Parsing Schemata, Springer-Verlag, Berlin, Heidelberg, pp. 253–288, 1997.
21. Sikkel, K., Salomaa, A. (Eds.) *Handbook of Formal Languages.* Springer-Verlag, Berlin, Heidelberg, 1997.
22. Šaloun, P. *Parallel Parsing of Strong LR Grammars.* Postgraduate Study Report, DCSE FEE CTU, Prague, 1996.

23. Soisalon-Soininen, E. and Tarhio, J. Looping LR Parsers. *Information Processing Letters*, 26:251–253, 1988.
24. Speckenmeyer, E. On Feedback Problems in Digraphs. In *Graph-Theoretic Concepts in Computer Science*, pp. 218–231. Springer-Verlag, Berlin, Heidelberg, 1989.
25. Tomita, M. *An Efficient Context-Free Parsing Algorithm for Natural Languages and Its Applications.* PhD Thesis, Carnegie-Mellon University, 1985.
26. Tomita, M. (Ed.) *Generalized LR Parsing.* Kluwer, Boston, London, 1991.
27. Wagner, T. A., and Graham, S. L. Incremental Analysis of Real Programming Languages. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 1997, pp. 31–43.