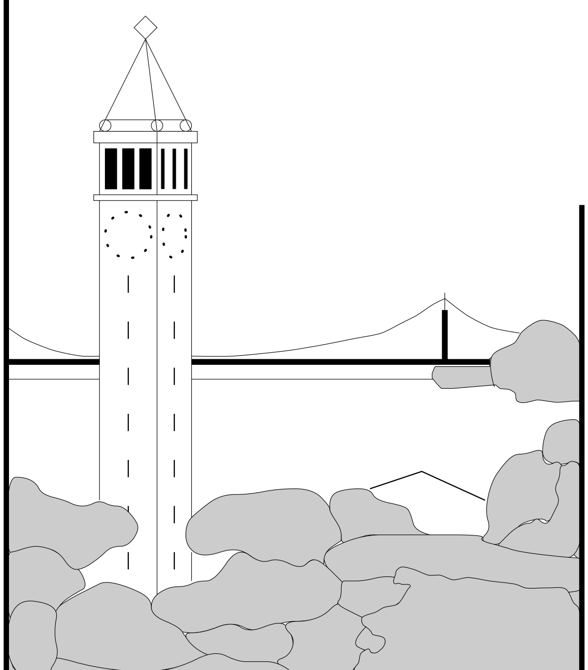
Elkhound: A Fast, Practical GLR Parser Generator

Scott McPeak
University of California, Berkeley
`smcpeak@cs.berkeley.edu`



Report No. UCB/CSD-2-1214

December 2002

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Elkhound: A Fast, Practical GLR Parser Generator

Scott McPeak
University of California, Berkeley
smcpeak@cs.berkeley.edu

December 2002

Abstract

Elkhound is a parser generator based on the Generalized LR (GLR) parsing algorithm. Because it uses GLR, Elkhound can parse with *any* context-free grammar, including those that are ambiguous or require unbounded lookahead. Due to a novel improvement to the GLR algorithm, wherein the parser can choose between GLR and ordinary LR on a token-by-token basis, Elkhound parsers are about as fast as LALR(1) parsers on deterministic portions of the input.

Unlike existing GLR implementations, Elkhound allows the user to associate arbitrary action code with reductions, and to directly manage subtree sharing and merging. These capabilities make Elkhound adaptable to a wide range of applications and memory management strategies.

To demonstrate Elkhound's effectiveness, we used it to build a parser for C++, a language notorious for being difficult to parse. Our C++ parser is small (3500 lines), efficient, maintainable, and elegantly handles even the most troublesome constructs—by delaying disambiguation until semantic analysis when necessary.

1 Introduction

The state of the practice in automated parsing has changed little since the introduction of YACC (Yet Another Compiler-Compiler), an LALR(1) parser generator, in 1975 [Joh75]. An LALR(1) parser is deterministic: at every point in the input, it must be able to decide which grammar rule to use, if any, utilizing only one token of lookahead [ASU86]. Not every context-free language has an LALR(1) grammar. For those that do, the process of modifying a grammar to conform to LALR(1) is difficult and time-consuming for the programmer, and it often involves obscuring or destroying conceptual structure in the grammar. Note that while any LR(k) grammar can be automatically transformed into an LR(1) grammar [HU79], the reduction actions *cannot* be automatically transformed.

The main alternative to a long battle with shift/reduce conflicts[1] is to abandon automatic parsing technology altogether. However, writing a parser by hand is tedious and expensive, and the resulting artifact is often difficult to modify to incorporate extensions. The Edison Design Group C++ Front End [EDG] includes such a hand-written parser for C++, and its size and complexity attest both to the difficulty of writing a parser without automation and the skill of EDG's engineers.

This paper describes Elkhound, a new parser generator intended to move the state of parsing practice beyond hand crafting and LALR(1). Based on the Generalized LR (GLR) parsing algorithm, Elkhound realizes a simple vision: describe your language with a context-free grammar, and the tool produces an efficient parser for that language.

1.1 Problems with LALR(1)

Given that parsers have been written using LALR(1) tools for twenty years, what is wrong with LALR(1)? The main problem is that it takes a lot of work to write an LALR(1) grammar for a given language. Resolving shift/reduce and reduce/reduce conflicts is typically a process of trial and error: stare at the parser generator's debug output, try to understand the conflict from the point of view of the parsing algorithm, and then modify the grammar. This

[1] A parse state in which the parser cannot decide whether to apply a grammar rule or consume more input is said to have a "shift/reduce" conflict. If the parser cannot decide *which* grammar rule to apply it has a "reduce/reduce" conflict.

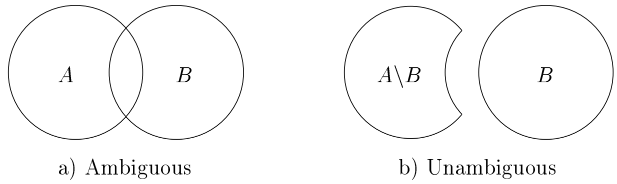


Figure 1: Recognizing a language difference.

process requires a good understanding of how LALR(1) works internally, and even with such knowledge it is by no means easy.

What's more, transforming a grammar into LALR(1) form usually destroys much of its original conceptual structure. Nonterminals cease to correspond to sub-languages, and instead come to represent states in the token by token decomposition of the input. Instead of describing the *language* to be parsed, the grammar describes the *process* used to parse it; it's more like a hand-crafted parsing program, but crammed into Backus-Naur Form. This is very unfortunate, since the grammar is the most important piece of the parsing specification.

LALR(1) also does not allow ambiguous grammars, yet they are certainly useful. Programming languages sometimes contain syntactic ambiguities when the languages for two constructs have a non-empty intersection, a situation suggested by Figure 1a. The language specification might disambiguate them with a phrase like "the resolution is to consider any construct that could possibly be a declaration a declaration."[2] Implementing any such resolution criteria with an unambiguous grammar would require describing the (asymmetric) *difference* between two sub-languages, which is messy at best and impossible at worst: the set of LALR(1) languages is not closed under subtraction. Figure 1b suggests resolving the ambiguity by classifying every string in $A \cap B$ as an instance of B, which would require changing the grammar for A to instead recognize $A \backslash B$.

Section 5.1 includes a dramatic application of an ambiguous grammar: since it is difficult in C++ to tell whether a given name refers to a type or a variable, but it makes a difference in how the surrounding syntax is parsed, we use a grammar in which names are parsed as *both* a type and a variable name. Any resulting ambiguities are then resolved during semantic analysis, a more appropriate phase than parsing in which to make the distinction.

1.2 Problems with GLR

The Generalized LR parsing algorithm [Lan74, Tom86] extends LR by effectively maintaining multiple parse stacks. Wherever ordinary LR faces a shift/reduce or reduce/reduce conflict, the GLR algorithm splits the stack to pursue all options in parallel. One way to view GLR is as a form of the Earley dynamic programming algorithm [Ear70], optimized for use with mostly-deterministic grammars. It can use any context-free grammar, including those that require unbounded lookahead or are ambiguous. Section 2 explains the GLR algorithm in more detail.

GLR addresses both of the main problems of LALR(1). However, while GLR is asymptotically as efficient as ordinary LR for deterministic input, GLR implementations are typically a factor of ten or more slower than their LR counterparts for grammars that are LALR(1). The poor performance is due to the overhead of maintaining a data structure more complicated than a simple stack, and of traversing that data structure to find reduction opportunities.

Additionally, most existing GLR parsers yield a parse tree (or a parse forest, in the case of ambiguous input) instead of executing user-specified actions at each reduction. They build such a tree because it allows the tool to control sharing and the representation of ambiguity. This is a problem because a parse tree is not a good data structure for subsequent language analysis; an abstract syntax tree (AST), designed with analysis in mind and constructed by reduction actions, is much better. Parse trees are very big, often consuming five to ten times as much memory as a corresponding AST. Further, parse trees have the wrong shape: later phases are forced to understand equivalent ways of saying the same thing, or to traverse past nonterminals which exist purely in service of the parser. Finally, they induce a tight coupling between the parsing grammar and later phases, since any change to the grammar forces adjustments to be made everywhere. Of course, an AST can be constructed by walking a parse tree, but then the effort of materializing the latter is wasted.

[2] [C++], Section 8.2, paragraph 1.

1.3 Advantages of Elkhound

The Elkhound parser generator is based on the GLR algorithm, so it inherits the benefits of unbounded lookahead and tolerance for ambiguity. Elkhound addresses GLR's relative performance problems by a novel enhancement which lets the parser use *both* GLR and ordinary LR, choosing between them at each token. For input fragments that exercise only the deterministic portions of the grammar, Elkhound parsers are as fast as conventional LALR(1) implementations: every conflict-free sub-language is parsed as fast as by a conventional LALR(1) implementation. As a result, a developer is free to take advantage of the convenience of GLR, and gradually work towards removing conflicts if desired for performance. Section 3.1 describes Elkhound's hybrid of GLR and LR in more detail.

Elkhound also supports the direct construction of abstract syntax trees, by executing arbitrary user action code with each reduction. However, this introduces two new challenges. First, if the grammar is nondeterministic, i.e. it requires more than one token of lookahead to decide how to parse, then some of the user's semantic values will be *shared* by the competing interpretations being built. Since sharing can have important semantic implications, Elkhound lets the user intervene in the process. The exposed interface is sufficiently powerful to implement any of the common schemes for resource management, including strict ownership (deep copying) and reference counting, though the defaults are intended for use with a garbage collector. In any case, this important decision is entirely in the user's hands.

Second, if the grammar is ambiguous, then the competing interpretations must be *merged* into a unified representation (otherwise the total forest size would be exponential in the number of ambiguities). Elkhound again defers to the user, who can specify for each ambiguous nonterminal in the grammar exactly how to merge semantic values.

These advantages make Elkhound fast, in that it creates parsers that are competitive with LALR(1) parsers, and practical, because it leaves the user in control of key decisions regarding management of semantic values. Elkhound is designed to make it easy to write a parser, by eliminating grammar restrictions, and easy to integrate that parser into a larger program, by avoiding constraints on its environment.

2 The GLR Parsing Algorithm

This presentation of the GLR [Lan74, Tom86] algorithm assumes the reader is acquainted with ordinary LR(1) or LALR(1) parsing. The needed concepts will be reviewed, but only briefly. For a more detailed introduction, see [ASU86].

2.1 LALR(1) Parsing Tables

Consider the following (ambiguous) expression grammar:

$$\begin{array}{lccl} 1. & S & \rightarrow & E \\ 2. & E & \rightarrow & i \\ 3. & E & \rightarrow & E + E \\ 4. & E & \rightarrow & E * E \end{array} \tag{EEi}$$

The first step in parser construction is to compute the item-set DFA (deterministic finite automaton), shown in Figure 2. The dot (•) indicates the parser's current position, while working through a given grammar production. The symbols after the comma are the lookahead tokens. Arcs from one state to another indicate transitions when shifting terminals, or after reducing nonterminals. Possible reductions have been circled.

The item-set DFA is computed during parse table generation, but the parsing decisions it records are traditionally encoded in two tables, the Action and Goto tables, shown in Figure 3. Entries of the form "Sn" mean it is possible to shift the column's terminal in the row's state, and that doing so leads to state n. Entries of the form "Rm" mean it is possible to reduce in the row's state when the column's terminal is the lookahead symbol, using grammar rule m. In the Goto table, the entries tell which state to go to, after reducing to the column's nonterminal.

Notice that there are four cells of the Action table with two possible actions. These are the shift/reduce conflicts, which in this case arise from the ambiguity in the grammar. While demonstrating the LR(1) algorithm, I will simply pick one of the two actions to use; later, the GLR algorithm will use both.

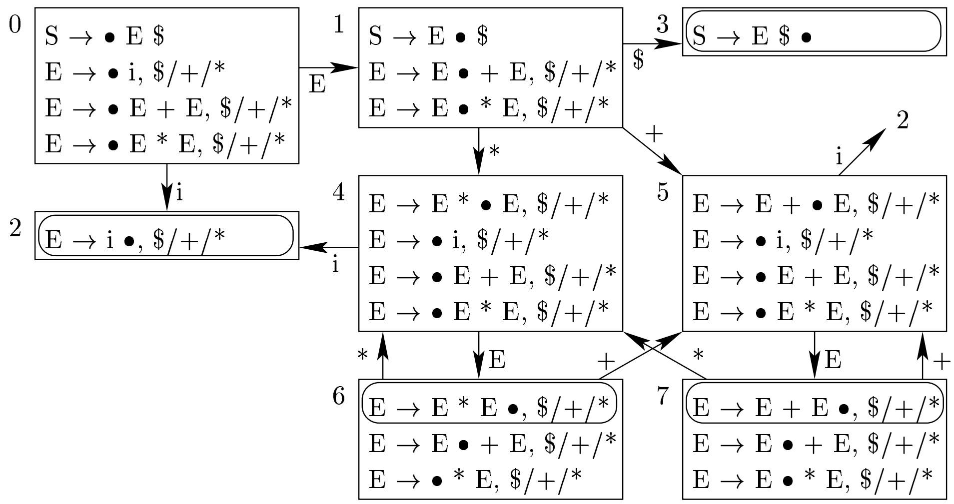


Figure 2: The item-set DFA for the *EEi* grammar.

	Action				Goto	
State	`$`	`i`	`+`	`*`	S	E
0		S2				1
1	S3		S5	S4		
2	R2		R2	R2		
3	⟵	accept using R1		⟶		
4		S2				6
5		S2				7
6	R4		S5 & R4	S4 & R4		
7	R3		S5 & R3	S4 & R3		

Figure 3: Action and Goto tables corresponding to Figure 2.

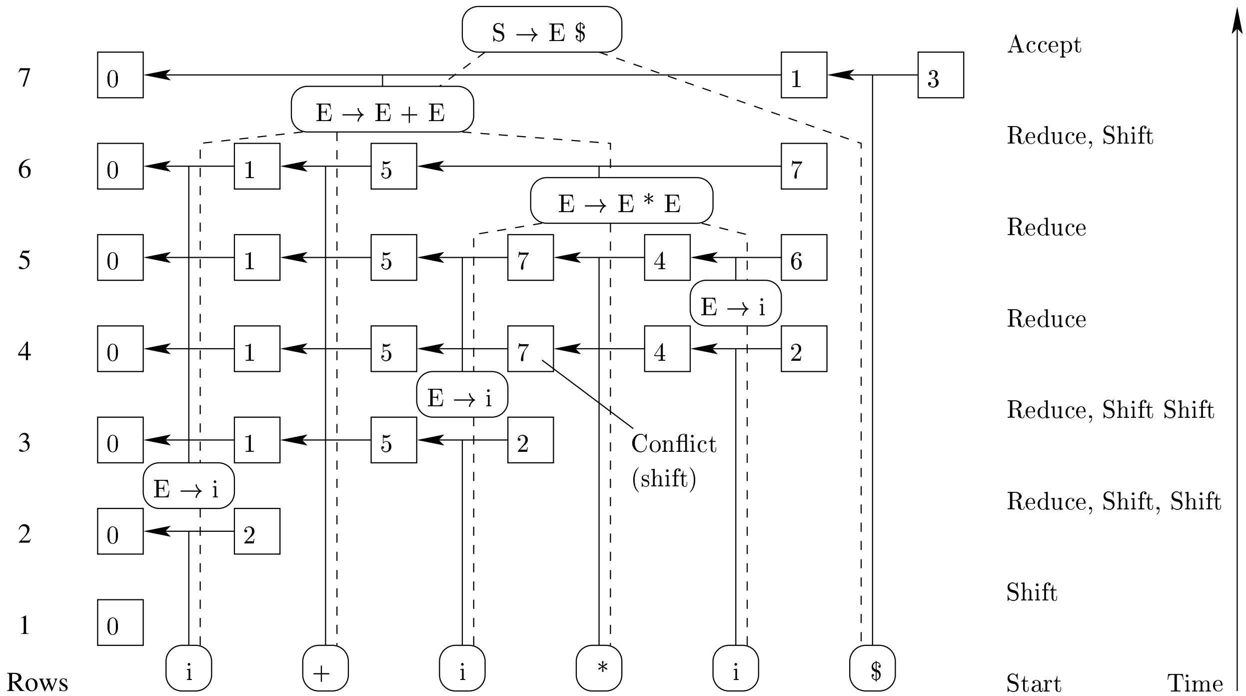


Figure 5: Behavior of an LR parser on input `i + i * i`. Square boxes are stack nodes, and rounded rectangles represents the result of running the user's reduction action code. Dotted lines show the flow of semantic values; they resemble parse tree edges.

2.2 The LALR(1) Parsing Algorithm

To establish context for the GLR algorithm, I first present the classic LALR(1) parsing algorithm.

During parsing, the algorithm maintains a stack of item-set state identifiers. Parsing proceeds by looking up the (state, lookahead token) pair in the Action table, and for reductions, consulting the Goto table to determine the next state. Figure 4 contains the details as pseudocode.[3]

Figure 5 diagrams the operation of the LR(1) algorithm on the input `i + i * i`. In this diagram, each row of square boxes is a snapshot of the stack at some point of execution. To the left of each state is the semantic value yielded by the reduction action (or the lexer) for the symbol reduced (or shifted) to arrive at that state. Time begins at the bottom, where the stack contains only state 0. This notation is unusual, but has a purpose: it can be generalized for use with GLR in subsequent sections. More traditional depictions of LR's behavior are harder to generalize.

The first action (row 2) is to shift the first symbol, after which the stack has two states, 0 and 2. Between them is the semantic value (lexeme) associated with the first symbol, indicated by the solid line extending down from between the states.

let <u>*stack*</u> of state identifiers contain the start state initially;
for each token of input <u>*t*</u> {
    if action(top(*stack*), *t*) = "reduce by <u>N</u> → <u>α</u>" {
        pop len(α) elements from *stack*;
        push goto(top(*stack*), N) onto *stack*;
    }
    else if action(top(*stack*), *t*) = "shift to <u>*dest*</u>" {
        push *dest* onto *stack*;
    }
    else {
        report parse error;
    }
}

Figure 4: The LR(1) parsing algorithm.

[3] The pseudocode notation uses **boldface** for keywords, *italics* for variable references, and <u>*underlined italics*</u> for binding introductions.

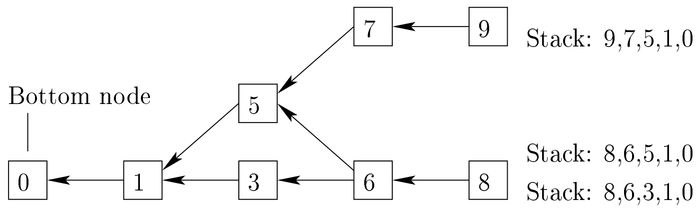


Figure 6: An example graph-structured stack. The nodes are labeled with their parse state number.

In row 3, the state containing 2 has been popped to execute the action associated with grammar rule $E \to i$. State 1 has been pushed, because $\text{goto}(\text{E}, \texttt{i}) = 1$. The semantic value between states 0 and 1 is not necessarily a parse tree node—it's whatever value the user's action returned. That action was given the semantic value of `i` as input, shown by the dashed line from `i` to the action bubble. Then, two more terminals are shifted, and four states are on the stack.

The diagram proceeds according to the LR algorithm in Figure 4, except when it encounters the `*` in state 7 on row 4. State 7 has two actions on `*`, a shift and a reduce. Here we show what happens when the algorithm chooses to shift, as it would under the conventional precedence of multiplication over addition.

Finally (row 7), the algorithm terminates when it enters state 3. At that time the last reduction action executes, and the result is passed out of the parser. By inspection, it should be clear that every semantic value is consumed exactly once, by a reduction action occurring later. This won't be true for the GLR algorithm.

2.3 The GLR Parsing Algorithm

As with LR parsing [ASU86], the GLR algorithm uses a parse stack and a finite control. The finite control dictates what parse action (shift or reduce) to take, based on what the next token is. But unlike LR, GLR's parse "stack" is not a stack at all: it is a graph which encodes all of the possible stacks that an LR parser could have. Each encoded stack is treated like a separate potential LR parser, and all stacks are processed in parallel, kept synchronized by always shifting a given token together.

The encoding of the GLR graph-structured stack (GSS) is simple. Some of the stack nodes are considered to be the top of their stack(s), and the algorithm keeps track of which are the top nodes. Every node has one or more directed edges to nodes below them in some stack, such that every finite path from a top node to the unique bottom node encodes a potential LR parse stack. Figure 6 shows one possible GSS and its encoded stacks. In the case of an ϵ-grammar [NF91], there may actually be a cycle in the graph and therefore an infinite number of paths. Elkhound can handle ϵ-grammars, but they are not considered further in this paper.

The GLR algorithm proceeds as follows: On each token, for each stack top, every enabled LR action is performed. There may be more than one enabled action, corresponding to a shift/reduce or reduce/reduce conflict in ordinary LR. A shift adds a new node at the top of some stack. A reduce also adds a new node, but depending on the length of the production's right-hand side, it might point to the top or into the middle of a stack. The latter case corresponds to the situation where LR would pop nodes off the stack; but the GLR algorithm cannot in general pop reduced nodes because it might *also* be possible to shift. It may be possible to reduce from a given node in more than one way if there is more than one path of the required length from that node, so the algorithm reduces along all such paths. If two stacks shift into the same state, or reduce into the same state, then the stack tops are merged into one node, encapsulating the ambiguous stack region. In Figure 6, the node with state 6 is such a merged node.

Figures 7 and 8 contain a pseudocode description of the algorithm. One unusual feature of the implementation shown here is that it uses a worklist of *reduction paths*, instead of a worklist of stack nodes. The reduction worklist serves two purposes. First, it avoids a subtle iteration sequencing problem that can cause some paths to be reduced more than once; this is explained in Section 2.4. Second, it makes the reduction ordering technique described in Section 4.2 possible, because all of the enabled reductions are available for inspection simultaneously. An earlier version of Elkhound contained implementations of both the stack worklist and reduction worklist algorithms, and their parsing performance was the same.

```
topmost: set of stack nodes;                               // global set of active parsers

pathQueue: queue of (path, production) pairs;              // worklist of enabled reductions

// toplevel GLR driver
glrParse(input: sequence of tokens) : returns the set of accepting stack nodes
{
    let start be a new stack node in the start state;
    topmost := { start };   pathQueue := ∅;
    for each token t in input {
        doReductions(t);
        doShifts(t);
    }

    remove nodes from topmost not in the accepting state;
    return topmost;
}

// perform all possible reductions; pathQueue is empty before and afterwards
doReductions(t)
{
    for each stack node current in topmost {               // seed the reduction worklist
        for each "reduce by N → α" in actions(current, t) {
            for each path p of length len(α) from current {
                enqueue (p, "N → α") into pathQueue;
            }
        }
    }

    while pathQueue is not empty {                         // process the reduction worklist until empty
        let (p, "N → α") = dequeue from pathQueue;
        reduceViaPath(p, "N → α", t);                      // execute reduction actions; see Figure 8
    }
}

// perform all possible shifts
doShifts(t)
{
    let prevTops = topmost;   topmost := ∅;
    for each stack node current in prevTops {
        if "shift to dest" is in actions(current, t) {
            if there is an existing stack node rightSib in topmost with state dest {
                addLink(current, rightSib, t);             // merge stack tops; see Figure 8
            }
            else {
                let rightSib = new stack node with state dest;
                insert rightSib into topmost;
                addLink(current, rightSib, t);
            }
        }
    }
}
```

Figure 7: GLR algorithm: Driver, shifts, and reduces.

```
// given a path p in the graph-structured stack, which corresponds to an instance of α, reduce it to N;
// t is the current lookahead token; there are three possible outcomes: a new stack node is created,
// a new stack link is added between two existing nodes, or a semantic value is merged
reduceViaPath(p, "N → α", t)
{
    let toPass[] =
        collect array of semantic values in links in p, where for each one we call dup(), storing the
        original value in toPass and dup's return value back into the link;
    let newSemanticValue = reductionAction("N → α", toPass);            // invoke user's reduction action

    let leftSib be the leftmost stack node in p;

    if there is already a node rightSib with state goto(leftSib, N) in topmost {
        if there is already a link from rightSib to leftSib {
            // merge the competing interpretations
            link.sval := merge(N, link.sval, newSemanticValue);         // invoke user's merge function
        }
        else {
            let link = addLink(leftSib, rightSib, newSemanticValue);

            // the new link might enable reductions in states whose reductions we've already expanded
            enqueueLimitedReductions(link, t);
        }
    }
    else {
        let rightSib = new stack node with state goto(leftSib, N);
        addLink(leftSib, rightSib, newSemanticValue);
        insert rightSib into topmost;
    }
}

addLink(leftSib, rightSib, semanticValue)
{
    let link = new link from rightSib to leftSib;
    link.sval := semanticValue;
}

// enqueue all reductions that use the newly-created link
enqueueLimitedReductions(link, t)
{
    for each stack node n in topmost {
        for each "reduce by N → α" in actions(n, t) {
            for each path p of length len(α) from n which uses link {
                enqueue (p, "N → α") into pathQueue;
            }
        }
    }
}
```

Figure 8: GLR algorithm: Execution of user actions, and link-specific enqueueing.

When the algorithm performs a reduction, it executes the user's action code. The result of an action is called a *semantic value*, and these values are stored on the links between the stack nodes.[4] When a reduction happens along a particular path, the semantic values stored on the links in that path are passed as arguments to the reduction action. If two different reductions lead to the same configuration of the top two stack nodes, i.e. the resulting stacks use the same final link, the algorithm merges their top-most semantic values. Each of the merged values corresponds to a different way of reducing some sequence of ground terminals to a particular nonterminal (the same nonterminal for both stacks). The merged value is then stored back into the link between the top two stack nodes, and participates in future reduction actions in the ordinary way.

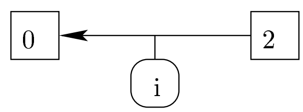
Deallocation of stack nodes is handled by reference counting; when a node is deallocated, all its sibling links are deallocated, and their semantic values are passed to `del()` (see Section 4.1).

While processing reductions in reduceSeveralPaths, the user's action code is run. The values returned by the actions are stored in the links which point between stack nodes. If a sibling link is added to an existing node, it could expose new opportunities to perform reductions, so a recursive pass through the *readyToShift* list is performed in that case. Figure 8 contains the details.

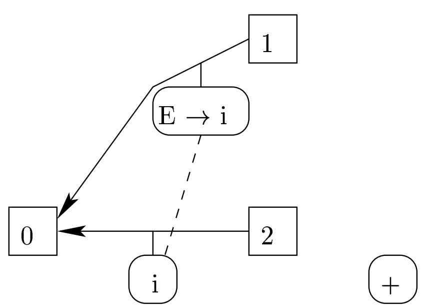
2.3.1 Example: Arithmetic Grammar

To illustrate the behavior of the GLR algorithm, I'll diagram its actions with the *EEi* grammar and the input `i + i * i`. As in Figure 5, nodes of the parse stack are in square boxes connected by solid lines, and nodes of the parse tree are in rounded boxes connected by dashed lines. A stack node's horizontal position denotes when, relative to the shifting of terminals, that stack node is created.

The first step is as in the LR algorithm: shift `i`. Now the parser is in state 2, with state 0 at the bottom of the stack. The sibling link between them refers to the lexeme for the `i` token, which is what will be passed to a reduction action later.

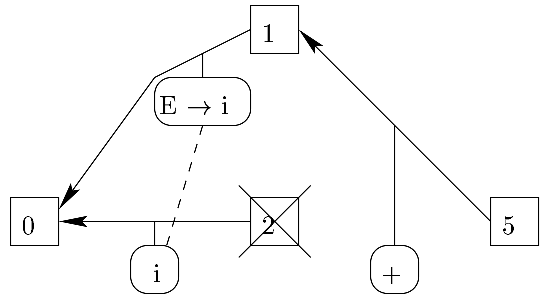


In state 2 on lookahead `+`, we can reduce, leading to state 1. The reduction action for $E \rightarrow i$ is executed, with the lexeme for `i` passed as an argument, and the result of the action is stored in the sibling link between states 1 and 2. However, the state 2 node is retained (for the moment), because it might be able to shift, too. (The basic GLR algorithm does not immediately notice that the parse tables disallow a shift in state 2. Section 3.1 describes an optimization that does.)

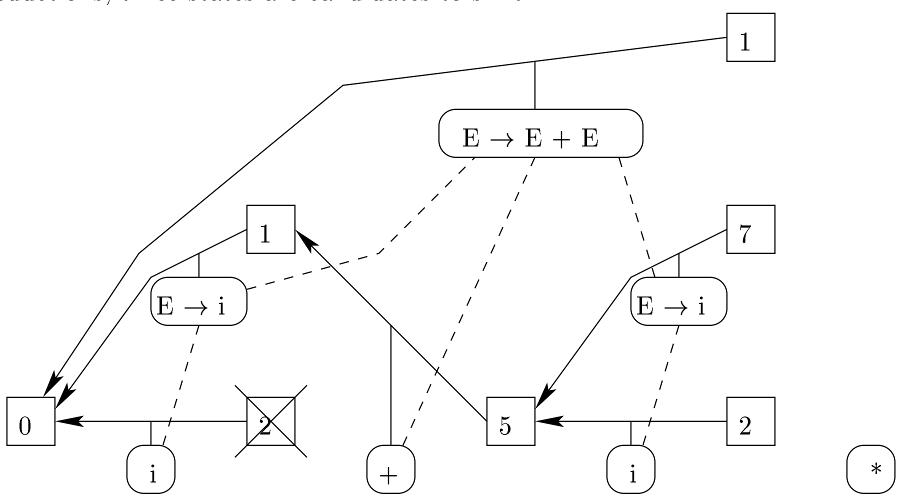


[4] It would be incorrect to store values in the stack nodes themselves, because a node at the top of multiple stacks must have a distinct semantic value for each stack.

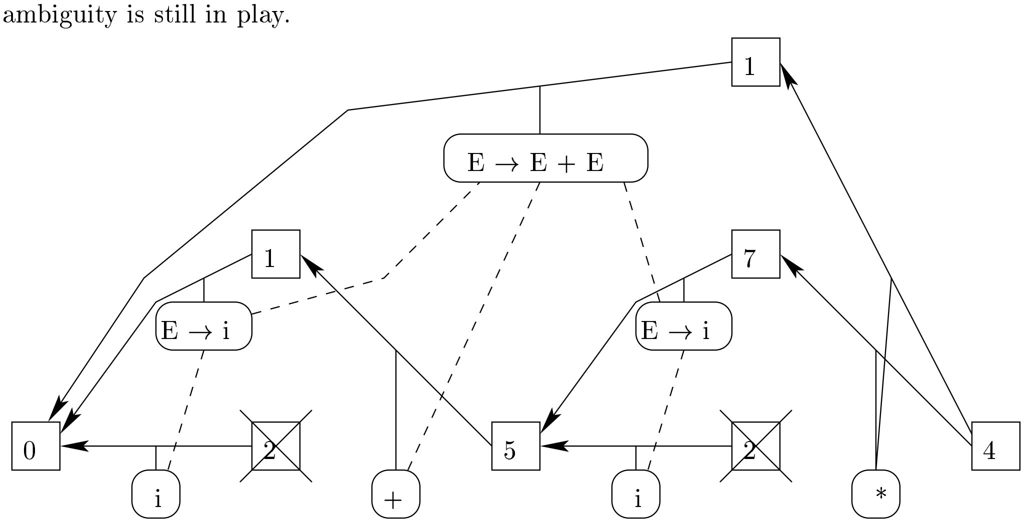
However, in the shift phase, only state 1 can shift the +, and doing so leads to state 5. The state 2 node cannot make progress, and is deallocated.



After shifting the next i, a sequence of two reductions are possible, leading in turn to states 7 and 1. After performing these reductions, three states are candidates to shift.

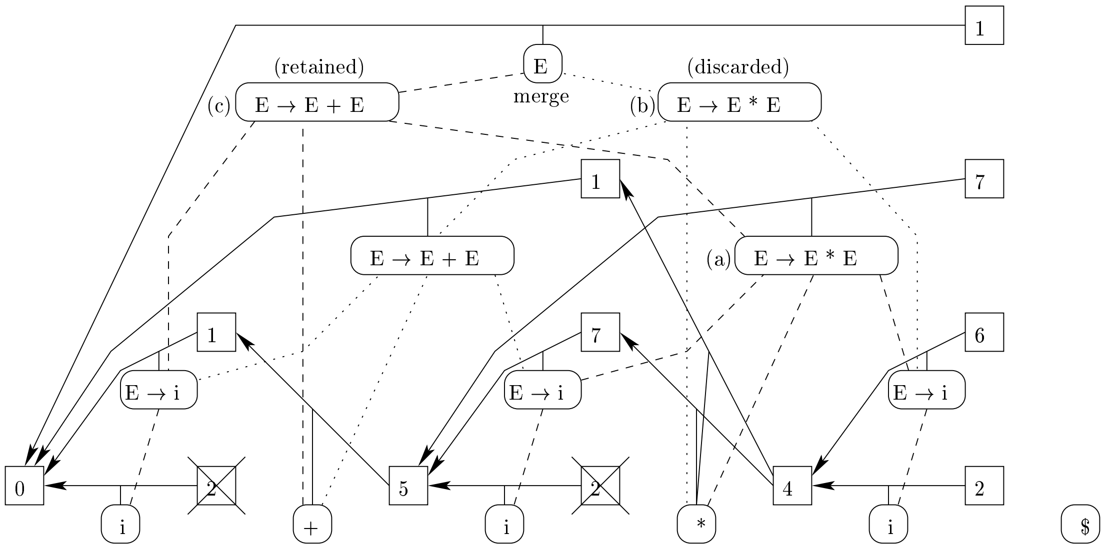


State 2 cannot shift *, but both states 7 and 1 can. This is where the ambiguity is first encountered: the prior step reduced from state 7, and now we shift as well. Since both shifts happen to lead to state 4, the forked stack tops are merged immediately. However, the node with 4 in it remembers it is the top of two stacks (it has two stack links), so the ambiguity is still in play.

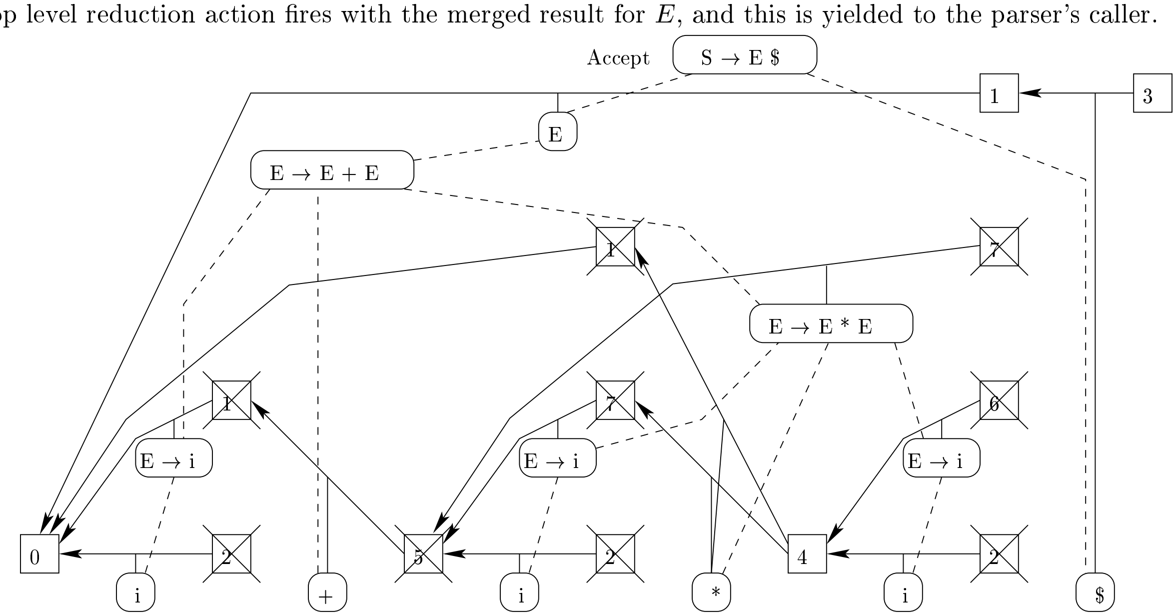


After shifting the final i, state 2 reduces to state 6. Then, state 6's potential reduction by $E \to E * E$ can be satisfied by two paths: 6,4,7,5 and 6,4,1,0. These two reduction instances lead to states 7 and 1 with semantic values labeled (a) and (b), respectively, when the E nonterminal is shifted.

Further, state 7 can reduce by $E \to E+E$, following path 7,5,1,0, which also leads to state 1 but with the semantic value labeled (c). Since there is already a link between the nodes for states 1 and 0, we have a true ambiguity: two different parse trees for the same sequence of ground terminals. To resolve this, we pass the two competing semantic values, (b) and (c), to the user-specified `merge` function associated with the E nonterminal. The `merge` function could keep both alternatives by constructing a tree node to point to them, but in this case we'll assume it decides to throw away (b), in accordance with conventional predence. The dotted lines connect the now-defunct subtree, and the dashed lines connect the retained subtree.



Finally, we shift the $. Only state 1 can shift this symbol, and doing so transitions to state 3, the accepting state. The top level reduction action fires with the merged result for E, and this is yielded to the parser's caller.



2.4 Bug Fix: Pathological Additional Link

The algorithm described in [Rek92] contains a subtle problem, such that a straightforward implementation would sometimes yield too many parse trees. Because it intermingles the enumeration of possible reductions with the processing of additional links created by those reductions (in DO-LIMITED-REDUCTIONS), if an additional link is added to the *same* state as is performing the reduction, then paths containing the new link may get processed twice: once by DO-LIMITED-REDUCTIONS, and once by the normal path enumeration. The duplicate processing leads to a spurious reduction and tree node.

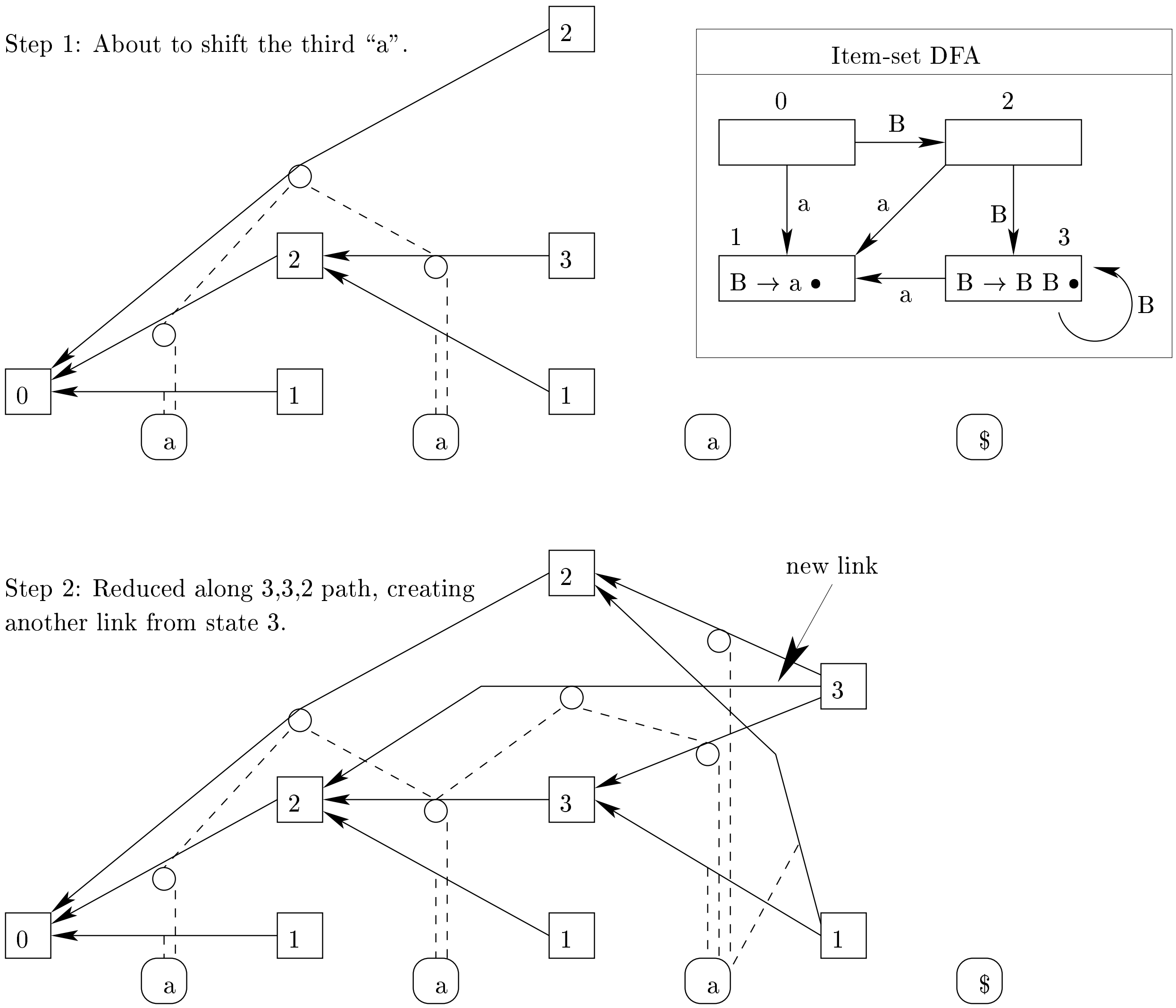


Figure 9: A GSS demonstrating a possible flaw in other GLR implementations.

Figure 9 demonstrates the problem on the input "aaa" with the grammar

$$B \quad \rightarrow \quad B\ B \ \mid \ a \qquad\qquad (BBa)$$

If, while performing reductions possible in state 3, the algorithm chooses to consider path 3,3,2 first, then it will add another link from state 3 to state 2. This will trigger a re-examination of all reduction paths, and the algorithm in [Rek92] will reduce along the newly-created 3,2,0 path. But then the outer state 3 reduction iteration will encounter the new link as well, and that path will be reduced again.

The fix is to enumerate all enabled reduction *paths* before doing any of them. Then, if a particular path adds a new link, the newly-enabled paths will *only* be processed by enqueueLimitedReductions (my equivalent to DO-LIMITED-REDUCTIONS). Elkhound's use of a reduction worklist achieves this automatically.

Note that the [Rek92] algorithm properly handles the case where the new link is added to a different state than is performing the reduction. In that case, either the new link extends from a state still "to do," in which case DO-LIMITED-REDUCTIONS ignores it, or else the state has already been marked "finished," so normal enumeration has completed.

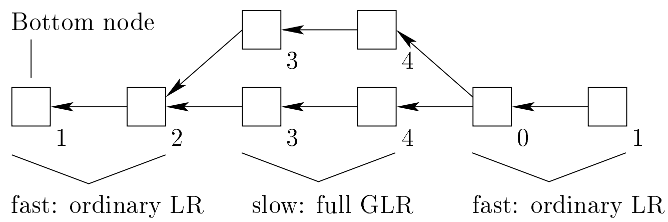


Figure 10: In this graph-structured stack, each node is labeled with its deterministic depth.

3 Performance Improvements

This section explains several enhancements to the basic GLR algorithm to make it faster. These enhancements are implemented in the Elkhound parser generator, but could reasonably be applied to any GLR implementation.

3.1 GLR/LR Hybrid

For most (programming) languages, the common case in the above scenario is that there is only one stack top and the parse action is unique. That is, for most of the input the ordinary LR algorithm would suffice.

It would be profitable to use LR when possible because performing reductions is much simpler (and therefore faster) with LR, and reductions account for the bulk of the time during parsing. The main cause for slower reductions with GLR is the need to interpret the graph-structured stack: following pointers between nodes, iteration over nodes' successors, and the extra mechanism to properly handle some special cases [Rek92, McP02] all add significant constant-factor overhead. Secondary causes include testing the node reference counts, and not popping and reusing node storage during reductions.

To exploit LR's faster reductions, one would like a way to decide when to use ordinary LR instead of the full GLR algorithm. Clearly, LR can only be used if there is a unique top stack node, and if the action for the current token at the top node's state is unambiguous. If that action is a shift, then a simple shift is performed: a new top node is created and the token's semantic value is stored on the new link.

However, if the action is a reduce, then we must check to see if the reduction can be performed more than once (via multiple paths), because if it can, then the GLR algorithm must be used. To enable this check, we modified the algorithm to keep track of each node's *deterministic depth*, defined to be the number of stack links that can be traversed before reaching a node with out-degree greater than one. The bottom node's depth is defined to be one. Any time the enabled reduction's right-hand side length (call it n) is less than or equal to the top node's deterministic depth, the reduction will only touch parts of the stack that are linear. Therefore a simple reduction can be performed: deallocate the top n nodes, and create in their place one new node, whose link will hold the reduction's semantic value.

Maintaining the deterministic depth is usually easy. When a node is created its depth is set to one more than that of its successor. When a second link is added to a node, its depth is reset to zero, and its reference count is inspected to see if there are other nodes with links to it. If there are, then the top nodes' depths are recomputed. Since this happens infrequently in practice, recomputation is cheaper than maintaining a list of pointers to predecessor nodes. Figure 10 shows an example parse stack annotated with the deterministic depths.

An important property of this scheme is that it ensures that the parsing performance of a given sub-language is independent of the context in which it is used. If we instead tried the simpler approach of using LR only when the stack is *entirely* linear, then (unreduced) ambiguity anywhere in the left context would slow down the parser. For example, suppose a C++ grammar contains two rules for function definitions, say, one for constructors and another for ordinary functions. If these rules have an ambiguity near the function name, that ambiguity will still be on the stack when the function body is parsed. By allowing ordinary LR to be used for the body despite the latent ambiguity, the parsing performance of the statement language is the same in any context. As a result, the effort spent removing conflicts from one sub-language is immediately beneficial, without having to chase down conceptually unrelated conflicts elsewhere. This also aids compositionality of grammar modules.

As shown in Section 6.1, the hybrid algorithm is about five times faster than the plain GLR algorithm for grammars that are LALR(1) or inputs that exercise only the LALR(1) fragment of a grammar.

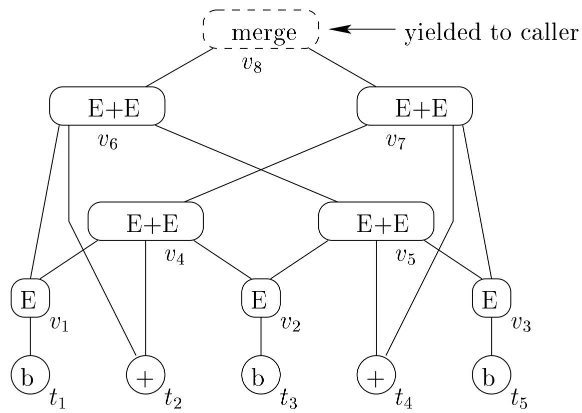


Figure 11: A parse forest for the *EEb* grammar.

3.2 Other Optimizations

The performance of the GLR algorithm depends heavily on the implementation of the graph-structured stack and the parse node worklists. In this section we briefly summarize the (empirically) most important optimizations applied to these data structures in Elkhound, beyond what is described in Section 3.1.

First, a naïve implementation of GLR does frequent allocation in its inner loops. All allocation of short-lived objects must be hoisted out of the loops, so the same memory is reused on each iteration. This avoids trips through the allocator, and improves locality.

Next, arrays (growable if necessary) should be preferred to linked lists whenever possible. In the inner loop, the algorithm walks the primary stack node worklist and the list of semantic values to pass to reductions. The extra indirection of a linked list would double parsing time.

Stack nodes contain a list of links to other stack nodes. However, most stack nodes contain only one such link, because most of the time the stack is approximately linear. Parsing time is cut in half by embedding the *first* link, including its associated semantic value reference, into the stack node object itself.

Finally, inlining and manual specialization are crucial to making the ordinary LR core fast. The LR core has to maintain the data structure invariants that the GLR core requires, but many of the manipulations it does are degenerate whenever LR can be used. For example, stack node reference counts can be statically predicted, so their updates can often be avoided; and stack nodes themselves can be directly reused during reductions (analogous to popping the parse stack in a conventional LR implementation).

The combined effect of the optimizations in this section is to save about a factor of eight in running time. Thus, the total effect of the techniques presented in Sections 3.1 and 3.2 is to make our implementation about forty times faster than a naïve implementation.

4 User-Specified Actions

The GLR algorithm's flexibility provides two basic challenges to any implementation that associates arbitrary user code with the reduction actions. First, while alternative parses are being pursued, semantic values are *shared* between the alternatives. This obviously entails questions about how to manage deallocation, but also raises the possibility of subtle interaction between the supposedly independent parses, if the actions have side effects. Second, if multiple parse trees can be constructed for a region of the input, the semantic values from the different interpretations have to be *merged* (otherwise the forest could be exponentially large).

As a running example, we use an ambiguous grammar for sums:

$$E \quad \rightarrow \quad E \;+\; E \;\mid\; b \qquad (EEb)$$

Figure 11 shows a parse forest for the input "b + b + b".

4.1 Sharing Subtrees

When a reduction action yields (produces as its return value for use by subsequent reductions) a semantic value such as v_1, it is stored in a link between two stack nodes. If that link is used for more than one subsequent reduction, then v_1 will be passed as an argument to more than one action; in Figure 11, it has been used in the creation of v_4 and v_6. If v_1 is a dynamically allocated object, then which reduction action takes responsibility for eventually deallocating it? One possibility is to use some form of garbage collection, another is to make a deep copy for each distinct use.

To allow for a range of memory management strategies, Elkhound allows the user to associate with each symbol (terminal and nonterminal) two functions, `dup()` and `del()`. `dup(`v`)` is called whenever v is passed to a reduction action, and its return value is stored back into the stack node link for use by the next action. In essence, the algorithm surrenders v to the user, and the user tells the algorithm what value to provide next time. When a node link containing value v is deallocated, `del(`v`)` is called. This happens when the last parser that could have potentially used v's link fails to make progress. Note that these rules also apply to semantic values associated with terminals, so t_2 and t_4 will be properly shared. As a special case, the calls to `dup()` and `del()` are omitted by the ordinary LR core, since semantic values are always yielded exactly once in that case.

Typical memory management strategies are easy to implement with this interface. For a garbage collector, `dup()` is the identity function and `del()` does nothing. To use reference counting, `dup()` increments the count and `del()` decrements it. Finally, for a strict ownership model, `dup(`v`)` makes a deep copy of v and `del()` recursively deallocates. This last strategy is fairly inefficient, so it should probably only be used in a grammar with at most occasional nondeterminism.

The fact that the algorithm stores `dup(`v`)` back in the stack link and yields the original v to the reduction action (as opposed to the other way around) is important because it lets the user control the lifetime of every value. The algorithm regards the semantic values stored in stack as borrowed from the user, and are returned as soon as possible. Were the algorithm to retain values indefinitely, it would limit the user's choices for semantic value representation.

As an application of the control afforded by this interface, consider a situation where the grammar author intends and expects that a given value will be used by at most one reduction action. With Elkhound's semantics for `dup`, a simple way to prevent a value from being used more than once is for `dup(`v`)` to return a special value which means "should not be used." This special value would be yielded to every reduction *after* the first, and actions could check for the special value, raising an error if they receive it. If the grammar indeed has the property that the value in question cannot be yielded more than once, then the error will never be raised. A particularly simple example is to use a `NULL` pointer as the special value, but of course this is up to the grammar author.

4.2 Merging Alternatives

If the grammar is ambiguous, then some inputs have more than one parse tree. In that case, semantic values representing the competing alternatives for the differing subtrees must be merged, so each nonterminal has an associated `merge()` function in the Elkhound API. For example, in Figure 11, semantic values v_6 and v_7 arise from different ways of parsing the same sequence of ground terminals, so the algorithm calls `merge(`v_6`,`v_7`)` and stores the return value v_8 back into the stack node link for use by future reduction actions.

Now, the user has at least three reasonable options in a `merge(`v_6`,`v_7`)` function: (1) pick one of the values to keep and discard the other one, (2) retain the ambiguity by creating some explicit representation of its presence, or (3) report an error due to unexpected input ambiguity. Option (3) is of course easy to do.

Unfortunately, options (1) and (2) don't always work in a naïve GLR implementation: depending on the order of reductions, by the time the `merge()` operation is performed, the value v_6 that would be replaced by `merge(`v_6`,`v_7`)` has already been used in another reduction, and thus the new `merge(`v_6`,`v_7`)` would be lost.

To illustrate the problem, consider the grammar *SAdB* and the GLR algorithm's activities while parsing "d":

$$\begin{array}{lcl} S & \rightarrow & A \\ A & \rightarrow & d \mid B \\ B & \rightarrow & d \end{array} \qquad (SAdB)$$

For reference, Figure 12a shows the states of the finite control. In Figure 12b, the d has been shifted, and the actions for $A \rightarrow d$ and $B \rightarrow d$ have been executed. But from there the algorithm can proceed in two ways because there are two top nodes, in states 2 and 3, which can reduce.

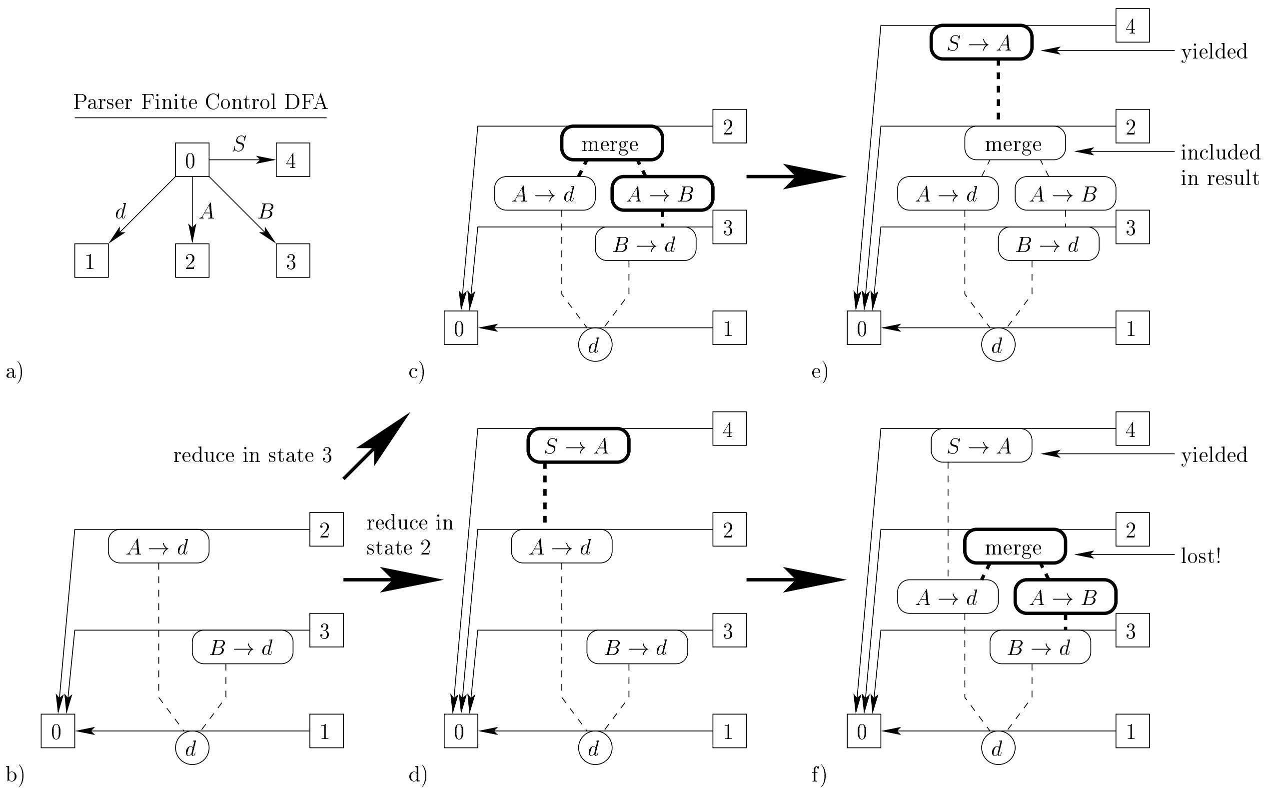


Figure 12: There are two possible reduction sequences for the *SAdB* grammar, depending on reduction order.

Figures 12c and 12e show the sequence of GLR stack configurations when the node in state 3 reduces first. First (Figure 12c), the action for $A \to B$ runs. Then, because this reduction leads to another stack configuration with state 2 on top of state 0, the algorithm merges the two semantic values for nonterminal A as well as the stacks themselves. Finally, (Figure 12e) the action for $S \to A$ runs. The semantic value corresponding to $A \to B$ participates in the final result because A was merged before it was passed to the action for $S \to A$.

On the other hand, Figures 12d and 12f show an alternative sequence, where state 2 reduces first. In that case, the action for $S \to A$ runs immediately (Figure 12d), and this is in fact the final result of the parse. Then (Figure 12f), the action for $A \to B$ and the `merge()` for A run, but nothing more is done with either value; the parser has already performed the reductions corresponding to state 2. The effect of one of the possible parses is lost.

This yield-then-merge problem can arise any time a grammar contains an ambiguous nonterminal N, and a production $M \to \alpha N \beta$ where $\beta \to^* \epsilon$ (α and β are arbitrary sequences of symbols, and ϵ is the empty string). Once the first semantic value for N is available, the reduction $M \to \alpha N \beta$ is enabled. If it runs before other possible values for N are computed, then the first value for N will have been yielded before being merged.

If the grammar is cyclic, meaning there is some nonterminal N such that $N \to^+ N$, then there may be *no* reduction order which avoids the problem. But if the grammar is acyclic, then for a given input there is always *some* order of reductions such that no value is ever yielded and then merged with another value: an acyclic grammar produces an acyclic parse forest, so an offline topological sort after parsing would find a good order.

Elkhound uses an online algorithm to find a reduction order that avoids yield-then-merge; this is where the reduction worklist presented in Section 2.3 comes into play. The reductions are maintained in a sorted order, such that:

- Rule 1. Reductions which span fewer tokens come first.
- Rule 2. If two reductions $A \to \alpha$ and $B \to \beta$ span the same tokens, then $A \to \alpha$ comes first if $B \to^+ A$.

This algorithm always avoids yield-then-merge if the grammar is acyclic.[5] To see why the algorithm works,

[5]If the grammar is cyclic then Rule 2 is not necessarily consistent, since it could be that both $B \to^+ A$ and $A \to^+ B$.

consider Figure 13, which shows the necessary conditions for yield-then-merge to happen: a given token sequence at the top of the parse stack can be interpreted as either α_1 or α_2, where $A \to \alpha_1$ and $A \to \alpha_2$ are possible reductions; and another reduction $B \to \beta A\gamma$ is also enabled, where $\gamma \to^* \epsilon$.

Now, if β cannot derive ϵ, then Rule 1 is sufficient to prevent the problem since both reductions to A will happen before the reduction to B. However, if $\beta \to^* \epsilon$ in this parse forest, then token span is not enough, and we turn to nonterminal derivability. Since $B \to^+ A$, Rule 2 specifies again that all reductions to A are performed before any to B. Since in both cases reductions to A preceed those to B, A is guaranteed to be completely merged before it is yielded.

Of course, this argument assumes that both $A \to \alpha_1$ and $A \to \alpha_2$ are present in the worklist at least as soon as $B \to \beta A\gamma$. Suppose (for purposes of contradiction) that $A \to \alpha_2$ did not get added to the worklist until after $B \to \beta A\gamma$ was added. By inspection, the algorthm presented in Section 2.3 adds reductions to the worklist as soon as their right-hand side constituents are in the GSS, so α_2 cannot consist entirely of terminals. Thus, $\alpha_2 = \delta C\theta$ for some C that was not present in the GSS. But all reductions to C should have happened before reductions to A: if δ didn't derive ϵ then Rule 1 applies, otherwise Rule 2 applies (clearly θ derived ϵ). Therefore C must have been present in the GSS by the time $A \to \alpha_1$ was reduced (itself a prerequisite for adding $B \to \beta A\gamma$), and we have a contradiction.

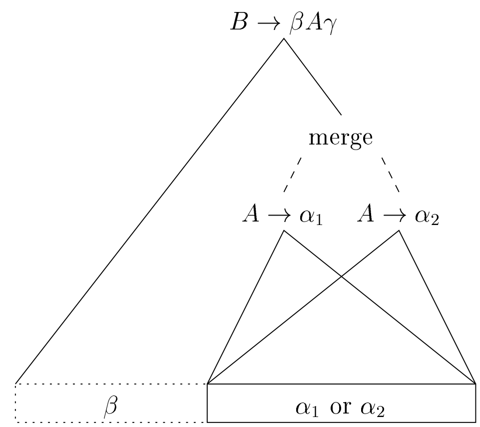


Figure 13: Necessary parse forest conditions to make yield-then-merge possible.

If the grammar is cyclic, then the algorithm above will not always avoid yield-then-merge. Fortunately, there is a simple design pattern for constructing potentially-ambiguous abstract syntax trees that works even if a node might be merged after being yielded. Given two AST nodes v_1 and v_2, `merge(`v_1`,`v_2`)` inserts v_2 into an "ambiguity" list inside v_1, and then returns v_1 itself. If v_1 and v_2 are pointers to objects with an `ambiguity` pointer, then the following `merge()` function (in Elkhound notation) does the required insertion:

```
merge(L,R) {      // prepend R to L's list
  R->ambiguity = L->ambiguity;
  L->ambiguity = R;
  return L;
}
```

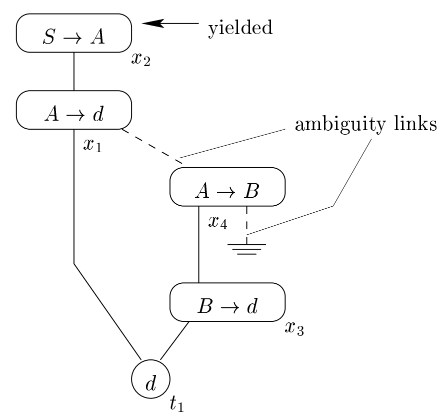


Figure 14: A parse forest for $SAdB$ in which ambiguities are represented using ambiguity links. The final forest is independent of reduction order.

Figure 14 shows a parse forest for $SAdB$ employing the `ambiguity` link solution. Even if x_2 is created before x_4, the forest accurately encodes all of the possible trees because `merge(`x_1`,`x_4`)` prepends x_4 to x_1's ambiguity list. It also returns x_1, to make sure that any addition reductions using the semantic value for A will see all of the interpretations.

It should be noted that traditional GLR implementations avoid the yield-then-merge problem by the way they construct parse trees. For example, the algorithm described in [Rek92] uses "Symbol" nodes, which are equivalent to (but use more memory than) the ambiguity links described above. What makes Elkhound different is the allowance for arbitrary user actions, which are not required to always retain ambiguities, nor incorporate subtrees by reference only. Such arbitrary actions are potentially sensitive to the order in which they are executed, and thus Elkhound must take care when choosing the reduction order.

4.3 Managing Side Effects

Beyond memory management, subtree sharing is a problem if reduction actions have side effects. The best policy is to avoid side effects whenever possible, which may mean unlearning some habits acquired while writing actions for LALR(1) tools. As a special case, it is safe for a side effect to modify the semantic value of one of its subtrees, *provided* that value is only yielded once. One way to verify that a value is yielded only once is for its `dup()` function to return `NULL`, as discussed in Section 4.1.

Side effects that modify global state like the symbol table are more difficult to implement correctly. It is possible to roll back changes in the `del()` function, but since other actions might run between the time the change is made and rolled back, this might not work in all cases. Developing a more general strategy for managing unavoidable global side effects is perhaps a topic for future work. The only clearly correct solution is to postpone all such side effects until after parsing, retaining any ambiguities this creates.

5 Case Study: A C++ Parser

To establish its real-world applicability, we put Elkhound to the test and wrote a C++ parser. This effort took one of the authors about three weeks. The final parser specification is about 3500 non-blank, non-comment lines, including the grammar, abstract syntax description and type checker. The grammar currently has 37 shift/reduce conflicts, 47 reduce/reduce conflicts and 8 ambiguous nonterminals.

This parser can parse and fully disambiguate most[6] of the C++ language, including templates. We used our implementation to parse Mozilla, a large (about 2 million lines) open-source web browser. Note that the type checker's primary purpose is to disambiguate, so it does not currently implement all of C++'s rules for type compatibility, though it would be straightforward to extend it to do so.

The C++ language definition includes several provisions that make parsing the language difficult. In the following sections we explain how we resolved these parsing difficulties using the mechanisms available in Elkhound.

5.1 Type Names versus Variable Names

The single most difficult task for a C or C++ parser is distinguishing type names (introduced via a `typedef`) from variable names. For example, the syntax "`(a)&(b)`" is the bitwise-and of `a` and `b` if `a` is the name of a variable, or a type-cast of the expression `&b` to type `a` if `a` is the name of a type. In C++ this task is even more difficult, since `a` might be a type name *whose first declaration occurs later in the file*: type declarations inside a class body are visible in all method definitions of that class, even those which appear textually before the declaration. For example:

```
int *a;                                // variable name (hidden)
class C {
  int f(int b) { return (a)&(b); }    // cast!
  typedef int a;                       // type name (visible)
};
```

The traditional solution, sometimes called the "lexer hack," is to add type names to the symbol table during parsing, and feed this information back into the lexical analyzer. Then, when the lexer yields a token to the parser, the lexer must first categorize the token as either a type name or a variable name. Type names adhere to the C scoping rules, so they can be hidden by other names and they eventually go out of scope. Thus the lexer is dependent on the implementation of scopes, a semantic concept.

In C, the lexer hack is non-ideal because of the tight coupling it induces among the lexer, parser, and semantic analyzer, but it is manageable. In C++, the scoping rules within classes make it considerably more difficult to implement: The parser must defer parsing of class method bodies until the entire class declaration has been analyzed, but this entails somehow saving the unparsed method token sequences and restarting the parser to parse them later.

However, with a GLR parser that can tolerate ambiguity, a much simpler and more elegant approach is possible: simply parse every name as *both* a type name and a variable name, and store both interpretations in the AST. During type checking, when the full AST and symbol table are available, one of the interpretations will fail because it has the wrong classification for a name. The type checker simply discards the failing interpretation, and the ambiguity is resolved. The scoping rules are easily handled at this stage, since the (possibly ambiguous) AST is available: make two passes over the class AST, where the first builds the class symbol table, skipping method bodies, and the second pass checks the method bodies.

[6]Namespaces ([C++] Section 7.3) and template partial specialization ([C++] Section 14.5.4) are not currently implemented because they are not needed to parse Mozilla. We foresee no new difficulties implementing these features.

5.2 Declarations versus Statements

Even when type names are identified, some syntax is ambiguous. For example, if `t` is the name of a type, the syntax "`t(a);`" could be either a declaration of a variable called `a` of type `t`, or an expression that constructs an instance of type `t` by calling `t`'s constructor and passing `a` as an argument to it. The language definition specifies ([C++], Section 6.8) that if some syntax can be a declaration, then it is a declaration; but establishing that it "can" be a declaration is not trivial.

The solution in this case is again to represent the ambiguity explicitly in the AST, and resolve it during type checking. If a statement can either be a declaration or an expression, then the declaration possibility is checked first. If the declaration is well-formed then that is the final interpretation. Otherwise the expression possibility is checked, and is used if it is well-formed. If neither interpretation is well-formed, then the two possible interpretations are reported to the user, along their respective diagnostic messages.

Not all disambiguation must be delayed until type checking. For example, an access declaration ([C++], Section 11.3) in a class body might look like "`B::x;`" where `B` is a base class. However, this could also be parsed as a member declaration with no declarators (if `B::x` is a type), taking advantage of the syntax that declares inner classes. To avoid this ambiguity we utilize Elkhound's `keep()` functionality, which can cancel a reduction. In this case the `keep()` function for class member declarations cancels the declaration if the type specifier does not introduce a new type name, and there are no declarators ([C++], Section 7, par. 3). For a conventional implementation to enforce this rule within the grammar, the grammar author would have to substantially modify the grammar, for example by splitting the declaration sub-language by cases on the form of the type specifier.

5.3 Function Declarators

In function parameter lists, the parameter names are optional. This creates an ambiguity between parentheses used to denote a function type, and parentheses used to control precedence:

```
typedef int x;          // 'x' is the only typedef
int f(int (x));         // what is this?
int g(int a(x b));      // parens mean function
int h(int (a));         // parens mean grouping
```

Does `f` accept an anonymous function-typed[7] argument, similar to `g`, or an integer-typed argument, similar to `h`? To paraphrase [C++] Section 8.2 par. 3, any type-name that occurs after a left-parenthesis which could either start a parameter list or serve to group a declarator is to be interpreted as naming the existing type (as opposed to introducing a new name). Therefore the parentheses mean "function," and `f` has the same type as `g`.

We resolve this ambiguity in the Elkhound C++ parser by recording explicitly the declarator grouping operator in the AST, so that during type checking it is possible to tell when we're in the context where ambiguity is possible. Then, in that context, the grouping interpretation is considered invalid if the declarator names an existing type. Figure 15 illustrates the ambiguity; the addition of the otherwise redundant "Grouping()" node to the AST is part of the solution. Whereas a conventional implementation would be forced to implement the lexer hack *and* recognize a difference between two languages, we use a simple tree pattern recognizer.

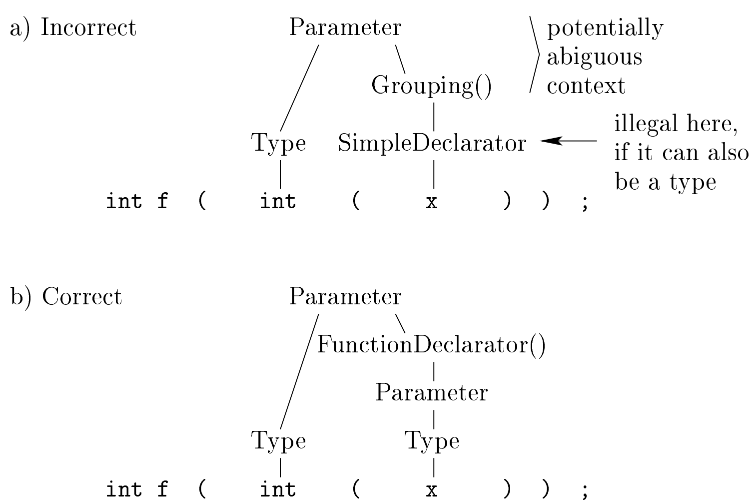


Figure 15: An ambiguous function declarator.

[7] Parameters with function type are automatically converted to pointer-to-function type ([C++], Section 8.3.5, par. 3).

5.4 Angle Brackets

Templates (also known as polymorphic classes, or generics) are allowed to have integer arguments. Template arguments are delimited by the angle brackets `<` and `>`, but these symbols also appear as operators in the expression language:

```
template <int n> class C { /*...*/ };
C<  3+4  > a;          // ok; same as C<7> a;
C<  3<4  > b;          // ok; same as C<1> b;
C<  3>4  > c;          // syntax error
C< (3>4) > d;          // ok; same as C<0> d;
```

The language definition specifies that there cannot be any unparenthesized greater-than operators in a template argument ([C++], Section 14.2, par. 3). Since this overloaded use of `>` rarely leads to a true ambiguity, our first implementation simply ignored the problem, effectively letting the parser look ahead to resolve each case. To handle the truly pathological cases like

```
new C< 3 > +4 > +5;         // correct parse is
((new C<3>) + 4) > (+5);    // <- like this
(new C< (3>(+4)) >) + 5;    // <- and not this
```

which have two valid parses according to our grammar, we modified the AST to record explicitly the use of parentheses to group expressions (similar to the treatment of declarators). Thus the type checker can detect an invalid unparenthesized `>` in a template argument with another simple pattern check.

A correct implementation in an LALR(1) setting would again require recognizing a difference between languages (as in Figure 1). In this case it would suffice to split the expression language into expressions with unparenthesized greater-than symbols and expressions without them. It is interesting to note that, rather than endure such a drastic change to the grammar, the authors of gcc-2.95.3 chose to use a precedence specification that works most of the time but is wrong in some cases: for example, gcc cannot parse the type "`C< 3&&4 >`". This is the dilemma all too often faced by the LALR(1) developer: sacrifice the grammar, or sacrifice correctness. Usually, the grammar is more important, as the gcc example attests.

5.5 Experience

The process of building the C++ parser with the GLR algorithm was both enjoyable and enlightening. By far the most difficult part was getting the environment lookup rules right. Dealing with parsing ambiguities was surprisingly simple, with essentially the same simple solution used in most places: represent the ambiguity in the AST, type check both, and discard the one that fails.

One of the pleasant surprises was the experience of debugging the grammar. A frequent problem was unexpected ambiguity, where the parser complains that there is no method defined to merge some nonterminal; by default, it prints a message and then discards one of the alternatives. Tracking down this problem is quite easy: Elkhound has an option to build and print a parse forest, so we would do just that, and study the resulting forest. This is ideal because we're debugging the grammar in terms of *ambiguity*, a concept directly related to the grammar, instead of debugging *conflicts*, a concept related to the parsing algorithm.

The other major unexpected benefit of using GLR is the predictability of the process. Whereas it's usually difficult to tell how far a given grammar is from being LALR(1), it's easy to count the kinds of ambiguous syntax in your test suite and estimate the work required to handle each one. Ambiguities tend to be local, non-interacting phenomena (again, unlike conflicts). Our work on the parser was able to proceed feature by feature, without having to reimplement one fragment because of a problem elsewhere.

6 Performance

In this section we compare the performance of Elkhound parsers to those of other parser generators, and also measure in detail the parsing performance of the C++ parser. The experiments were performed on a 1GHz AMD Athlon running Linux. We report the median of five trials.

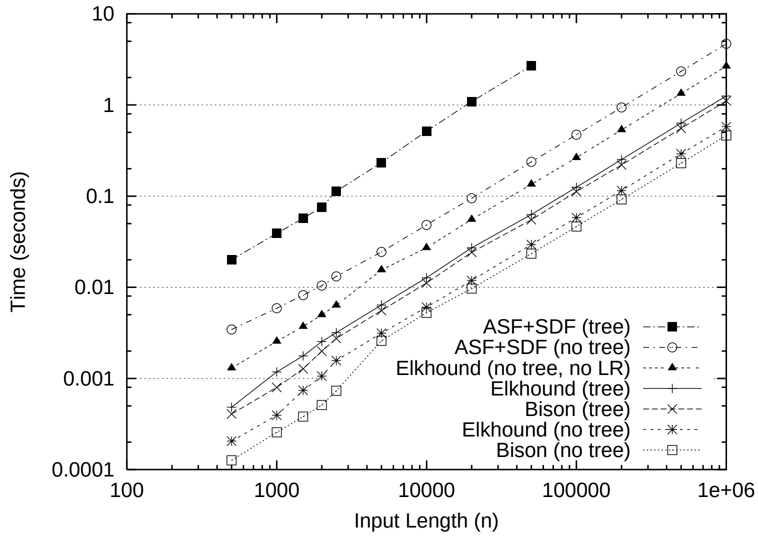


Figure 16: Parser performance on the *EFa* grammar, on a log-log scale. Input string is $a(+a)^n$.

6.1 Comparison to Other Parser Generators

For comparison with a conventional LALR(1) implementation, we compare Elkhound to Bison, version 1.28 [DS99]. Bison associates user action code with each reduction, and generates a parser written in C.

For comparison with an existing GLR implementation, we used the ASF+SDF Meta-Environment [HHKR89]. The Meta-Environment contains a variety of language-processing tools, among which is a scannerless GLR parser [Vis97] that generates parse trees. The parser can be run as a recognizer, and in this mode does not build trees. We used the Meta-Environment bundle version 1.1.1, which contains version 3.7 of the SGLR component. This package is written in C.

To measure the speed of Elkhound's ordinary LR parser, we measured its performance against Bison and ASF+SDF on this LALR(1) grammar:

$$\begin{array}{rcl} E & \rightarrow & E \ + \ F \ \mid \ F \\ F & \rightarrow & a \ \mid \ (\ E \ ) \end{array} \qquad (EFa)$$

We measured each tool both with and without executing tree-building actions. As shown in Figure 16, the Elkhound LR parser core is only a few percent slower than Bison, whereas ASF+SDF is a factor of ten slower than both. When Elkhound's LR parser (Section 3.1) is disabled, parsing slows down by a factor of five. This validates the hybrid design: the overhead of choosing between LR and GLR is almost negligible, and the speed improvement when LR is used is substantial.

We also measured performance on the highly ambiguous *EEb* grammar, reproduced here:

$$E \ \rightarrow \ E \ + \ E \ \mid \ b \qquad (EEb)$$

This grammar generates the language described by the regular expression $b(+b)^*$, and for the input string $b(+b)^n$ the number of parses $C(n)$ is exponential:

$$\begin{array}{rcl} C(n) & = & \displaystyle\sum_{m=0}^{n-1} C(m) \cdot C(n-m-1) \quad = \quad \binom{2n}{n}\frac{1}{n+1} \\ C(0) & = & 1 \end{array}$$

As shown in Figure 17, Elkhound's performance is very similar to that of ASF+SDF, when neither builds trees. Both require time approximately quartic in the input size; for grammars with such a high degree of ambiguity, GLR is often slower than the Earley algorithm [Ear70]. Since ASF+SDF apparently tries to materialize all of the parse trees separately without sharing subtrees, it cannot build trees for inputs with $n > 10$, whereas Elkhound is only a factor of two slower when building trees. This illustrates one of the drawbacks of tools that build trees: they sometimes have unexpected limitations.

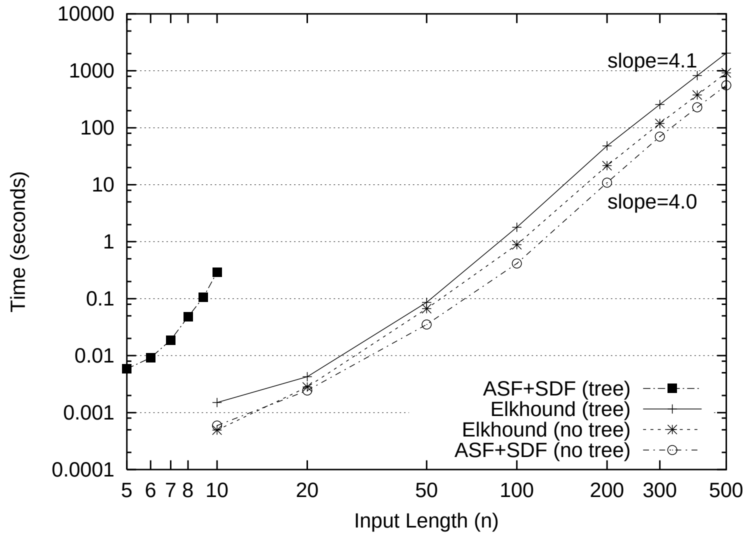


Figure 17: Parser performance on the *EEb* grammar, on a log-log scale. Input string is $b(+b)^n$.

Table 1: C++ Parser Performance.

Preprocessed File Name	Lines	Parse ms	No LR ms (×)	Tcheck ms	g++ ms (×)
nsUnicodeToTeXCMRt1.i	9537	16	36 (2.25)	50	60 (1.10)
nsAtomTable.i	19369	104	179 (1.72)	296	270 (1.48)
nsCLiveconnectFactory.i	24055	80	167 (2.09)	273	250 (1.41)
nsSOAPPropertyBag.i	26807	173	298 (2.30)	418	460 (1.28)
nsMsgServiceProvider.i	39215	209	378 (1.81)	545	560 (1.35)
nsHTMLEditRules.i	49566	495	827 (1.67)	934	1140 (1.25)

6.2 C++ Parser Performance

To test the C++ parser, and measure its performance, we used it to parse Mozilla 1.0. Mozilla has about 2000 source modules in its Linux configuration, averaging about 30000 preprocessed lines each. Our C++ parser can parse and fully disambiguate all of the modules except two that use gcc extensions not currently implemented.

We selected six of Mozilla's modules at random to measure and report parsing performance. Table 1 shows several measurements for each file. Parsing time is reported in milliseconds. "No LR" is parse time when the LR hybrid mechanism (Section 3.1) is disabled, and (×) is the ratio of No LR to Parse. Tcheck is the time for the type checker to run; Parse + Tcheck is the total time to parse and disambiguate. g++ is the time for gcc-2.95.3 to parse the code, as measured by its internal `parse_time` instrumentation,[8] and (×) is the ratio of Elkhound parse time to g++ parse time.

The No LR measurements show that, while the GLR/LR hybrid technique is certainly beneficial, saving about a factor of two, it is not as effective for the C++ grammar as it is for a completely deterministic grammar such as *EFa*, where it saves a factor of five. Of course, the reason is the C++ parser cannot use the LR parser all the time; on the average it can use it only about 70% of the time. Presumably if we spent more effort modifying the grammar to remove conflicts, the performance could be improved, but for now there appears to be no need to do so.

Remarkably, the Elkhound C++ parser is typically only 30–40% slower than gcc's C++ parser. As an engineering trade-off, this meager performance difference seems to be more than compensated for by the vastly simpler design of the former parser.

[8] We modified gcc's source to enable `parse_time`, a measurement which does not include any code generation activities. It does include type checking, however, because gcc does such checking at the same time as it builds its internal representation of the code.

7 Related Work

In "Current Parsing Techniques in Software Renovation Considered Harmful" [vdBSV98], the case for GLR is argued on the basis of compositionality and other important grammar closure properties. That argument in part inspired work on Elkhound.

The ASF+SDF Meta-Environment [HHKR89] contains a robust, efficient GLR parser. However, it suffers from the problems mentioned in Section 1.2. The Harmonia development environment [Bos01] makes novel use of an incremental GLR parser. There is previous work to make the GLR algorithm faster [AHJM01, ACV97]; incorporating such improvements into Elkhound is possible future work. Paul Hilfinger recently added support for GLR parsing to version 1.5 of Bison. Work is underway to compare the performance of the GLR parsers in Elkhound and Bison.

Several strategies for disambiguation in ASF+SDF are discussed in [vdBSVV02], including `reject` and `prefer` directives which are similar to Elkhound's `keep()` and `merge()` option (1), respectively. Elkhound's mechanisms are more expressive because user writes code to specify disambiguation criteria, while the declarative `reject` and `prefer` are possibly more efficient if the parser generator can figure out how to apply them earlier in the parsing process.

Besides GLR, there are several other parsing algorithms that can handle nondeterminism or ambiguity. The oldest of these is the Earley dynamic programming algorithm [Ear70], with running time $\Omega(n^2)$ and $O(n^3)$. The ACCENT parser generator [Sch00] uses a combination of the LL and Earley algorithms.

Some tools, such as BtYacc and ANTLR, use extensions to LL and LR to achieve greater lookahead. These extensions are neither as efficient nor as general as GLR, particularly with respect to tolerance for ambiguity. Similarly, one can use higher-order combinators in functional programming languages [Hut92] to conveniently describe backtracking LL parsers capable of unbounded lookahead and ambiguity representation, but they consume exponential time and space in the worst case.

8 Future Work

The most important feature missing from Elkhound is error diagnosis and recovery; the current implementation simply stops and prints a cryptic message when there is a parse error. I plan to implement the Burke-Fisher [BF87] error diagnosis scheme.

As many language research projects are now being implemented in dialects of ML, it would be quite convenient if Elkhound's back-end could generate and use actions written in ML. The parser never directly inspects the user's semantic values, so they could be just as easily handles to ML or Java data structures as pointers to C++ objects. If Elkhound were in general retargetable to different action languages it could be of use to a wider audience.

9 Conclusion

The Elkhound parser generator improves upon existing GLR technology in two ways. First, it uses a hybrid parsing algorithm able to switch between ordinary LR and full GLR on the fly, for each token. This lets the grammar developer choose whether to resolve some conflicts to get LR-like speed, or leave them for GLR-like readability. Second, Elkhound exposes a full-featured interface for controlling sharing and merging of semantic values, without building intermediate data structures such as a parse tree. This ensures scalability and predictable performance.

This paper also tries to articulate the benefits of GLR parsing. Developing a parser using Elkhound is much easier than it is with LALR(1) tools. It does not require intimate knowledge of any parsing algorithm. It also does not require the ability to fully disambiguate input during parsing; an Elkhound parser can extract exactly as much structure as is convenient during parsing. With these advantages, we were able to use this technology to build a C++ parser in less than one person-month.

The time has come for a new breed of parsing tools to displace the quirky and frustrating LALR(1) algorithm. The implementation described in this paper demonstrates that moving beyond LALR(1) does not require sacrificing performance.

Elkhound and its C++ parser have been released under an open-source (BSD) license, and are available at `http://www.cs.berkeley.edu/~smcpeak/elkhound` .

References

[ACV97] Miguel A. Alonso, David Cabrero, and Manuel Vilares. Construction of efficient generalized LR parsers. In *WIA: International Workshop on Implementing Automata, LNCS*. Springer-Verlag, 1997.

[AHJM01] John Aycock, R. Nigel Horspool, Jan Janoušek, and Bořivoj Melichar. Even faster generalized LR parsing. *ACTAINF: Acta Informatica*, 37(9):633–651, 2001.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BF87] Michael G. Burke and Gerald A. Fisher. A practical method for LR and LL syntactic error diagnosis. *TOPLAS*, 9(2):164–197, April 1987.

[Bos01] Marat Boshernitsan. Harmonia: A flexible framework for constructing interactive language-based programming tools. Technical Report CSD-01-1149, University of California, Berkeley, June 2001.

[DS99] Charles Donnelly and Richard M. Stallman. *Bison: the YACC-compatible Parser Generator, Bison Version 1.28.* Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, January 1999.

[Ear70] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[EDG] Edison Design Group. `http://www.edg.com`.

[HHKR89] Jan Heering, Paul R. H. Hendriks, Paul Klint, and Jan Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, November 1989.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[Hut92] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992.

[C++] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++.* International Organization for Standardization, Geneva, Switzerland, September 1998.

[Joh75] Stephen C. Johnson. YACC - yet another compiler compiler. Technical Report 32, Bell Telephone Laboratories, 1975.

[Lan74] Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Automata, Languages and Programming*, volume 14 of *Lecture Notes in Computer Science*, pages 255–269. Springer, 1974.

[McP02] Scott McPeak. Elkhound: A fast, efficient GLR parser generator. Technical Report CSD-02-1214, University of California, Berkeley, December 2002. `http://www.cs.berkeley.edu/~smcpeak/elkhound`.

[NF91] Rahman Nozohoor-Farshi. GLR parsing for ϵ-grammars. In Masaru Tomita, editor, *Generalized LR Parsing*, pages 61–75. Kluwer, 1991.

[Rek92] Jan Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, 1992.

[Sch00] Friedrich Wilhelm Schröer. The ACCENT compiler compiler, introduction and reference. Report 101, German National Research Center for Information Technology, July 2000.

[Tom86] Masaru Tomita. *Efficient Parsing for Natural Language.* Int. Series in Engineering and Computer Science. Kluwer, 1986.

[vdBSV98] Mark van den Brand, Alex Sellink, and Chris Verhoef. Current parsing techniques in software renovation considered harmful. In *Proceedings of the 6th International Workshop on Program Comprehension*, pages 108–117, Ischia, Italy, 1998.

[vdBSVV02] Mark van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Norspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158, Grenoble, France, April 2002. Springer.

[Vis97] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, University of Amsterdam, 1997.