Elkhound: A Fast, Practical GLR Parser Generator

Scott McPeak and George C. Necula *

University of California, Berkeley
{smcpeak,necula}@cs.berkeley.edu

Abstract. The Generalized LR (GLR) parsing algorithm is attractive for use in parsing programming languages because it is asymptotically efficient for typical grammars, and can parse with any context-free grammar, including ambiguous grammars. However, adoption of GLR has been slowed by high constant-factor overheads and the lack of a general, user-defined action interface.

In this paper we present algorithmic and implementation enhancements to GLR to solve these problems. First, we present a hybrid algorithm that chooses between GLR and ordinary LR on a token-by-token basis, thus achieving competitive performance for determinstic input fragments. Second, we describe a design for an action interface and a new worklist algorithm that can guarantee bottom-up execution of actions for acyclic grammars. These ideas are implemented in the Elkhound GLR parser generator.

To demonstrate the effectiveness of these techniques, we describe our experience using Elkhound to write a parser for C++, a language notorious for being difficult to parse. Our C++ parser is small (3500 lines), efficient and maintainable, employing a range of disambiguation strategies.

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 [1]. 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 [2]. Not every context-free language has an LALR(1) grammar. Even for those that do, the process of modifying a grammar to conform to LALR(1) is difficult and time-consuming for the programmer, and this transformation often 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

* This research was supported in part by the National Science Foundation Career Grants No. CCR-9875171, No. CCR-0081588, No. CCR-0085949 and No. CCR-0326577, and ITR Grants No. CCR-0085949 and No. CCR-0081588, and gifts from Microsoft Research.

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.

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, especially those of interest in a research setting. The Edison Design Group C++ Front End 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 improvements and enhancements to the Generalized LR (GLR) parsing algorithm that make its performance competitive with LALR(1) parsers, and its programming interface flexible enough for use in a variety of real-world scenarios. These enhancements are demonstrated in the Elkhound parser generator.

1.1 The State of GLR

The Generalized LR parsing algorithm [3–5] 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 [6], optimized for use with mostly deterministic[2] 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 is a much more convenient technology to use than LALR(1). It does not impose constraints on the grammar, which allows rapid prototyping. Further, unlike the Earley algorithm, its performance improves as the grammar approaches being deterministic, so a prototype grammar can be evolved into an efficient parsing grammar incrementally.

However, while GLR is asymptotically as efficient as ordinary LR for deterministic input, even mature GLR implementations such as ASF+SDF [7] are typically a factor of ten or more slower than their LALR counterparts for deterministic grammars. Consequently, users are reluctant to use GLR. The poor performance is due to the overhead of maintaining a data structure more complicated than a simple stack, and traversing that data structure to find reduction opportunities.

Existing GLR parsers build 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. However, because the user cannot control the representation of the parser's output, applicability is limited. Most commonly, the

[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.

[2] In this context, a grammar is deterministic if it is LALR(1).

user of a parser would like to build an abstract syntax tree (AST) instead of working directly with a parse tree. Or, the analysis task might be simple enough to do during parsing itself. Traversing a parse tree afterwards to simulate on-line reduction actions would once again incur the kinds of performance problems that inhibit adoption.

1.2 Contributions

First, to improve parsing performance on deterministic fragments of grammars, we present an enhancement that allows the parser to dynamically switch between GLR and ordinary LALR(1) at the token level of granularity. The information needed to soundly decide when LALR is sufficient is easily maintained in most cases. Even though it must operate on the GLR data structures, the LALR parser core is much faster because it can make much stronger assumptions about their shape and about the possible sequences of upcoming parsing actions. With this improvement, parsing performance on deterministic grammars is within 10% of good LALR implementations such as Bison [8].

Second, we present a design for a set of user-specified action handlers that gives the user total control over representation decisions, including subtree sharing. Further, we present a crucial modification to the GLR algorithm that ensures the action handlers are executed in a bottom-up order (for acyclic grammars); the original GLR algorithm cannot make this guarantee.

2 The GLR Parsing Algorithm

Since the reader may be unfamiliar with the GLR parsing algorithm [3–5], this section contains a brief explanation of the key points. For a more thorough description see [9], an expanded version of this paper.

As with LR[3] parsing [10, 2], 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, and the stack summarizes the left context as a sequence of finite control state numbers. But unlike LR, GLR's parse "stack" is not a stack at all: it is a graph which encodes all of the possible stack configurations 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. Every node has one or more directed edges to nodes below it in some stack, such that every finite path from a top node to the unique bottom node encodes a potential LR parse stack. Figure 1 shows one possible GSS and its encoded stacks. In the case of an ϵ-grammar [11], there may actually be a cycle in the graph and therefore

[3] GLR is built on top of the finite control of an LR parser, such as LR(0), SLR(1), LALR(1) or even full LR(1). Elkhound uses LALR(1). In this paper we use the term LR to refer nonspecifically to any of these.

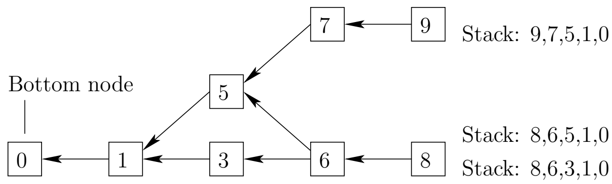


Fig. 1. An example graph-structured stack. Each node is labeled with its parse state number.

an infinite number of paths; Elkhound can handle ϵ-grammars, but they are not further considered 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 node. 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. If there is more than one path of the required length from the origin node, the algorithm reduces along all such paths. If two stacks shift or reduce into the same state, then the stack tops are merged into one node. In Figure 1, the node with state 6 is such a merged node.

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 GSS 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): an ambiguity. 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.

3 GLR/LR Hybrid

When using a parsing grammar for a programming language, the common case for the GLR algorithm is to have only one top stack node, and one possible parse action. That is, for most of the input, the ordinary LR algorithm would suffice. For our C++ grammar about 70% of the parse actions fit this description.

[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.

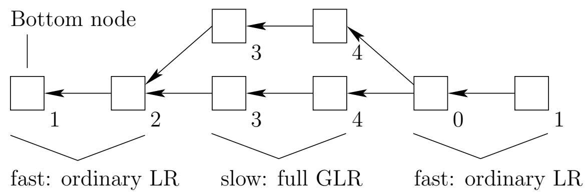


Fig. 2. In this graph-structured stack, each node is labeled with its deterministic depth.

It would therefore be profitable to use LR when possible because performing reductions is much simpler (and therefore faster) with LR, and reductions account for the bulk (about 80% for the C++ grammar) of the actions 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 [5, 9] 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, we need a way to predict when using LR will be equivalent to using GLR. 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 we can do a simple LR shift: 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 make sure that all possible reductions are considered. 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. Figure 2 shows an example. 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 (in-degree and out-degree at most one). 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 outgoing link is added to a node, its depth is reset to zero. When a node's depth is reset, if any nodes point at it, their depths are also potentially affected and so must be recomputed. This happens rarely, about 1 in every 10,000 parse actions for the C++ grammar, and can be done efficiently with a topological sort (details omitted due to space constraints).

An important property of the hybrid 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.

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.

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. 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*.

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

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

Figure 3 shows a parse forest for the input "$b + b + b$".

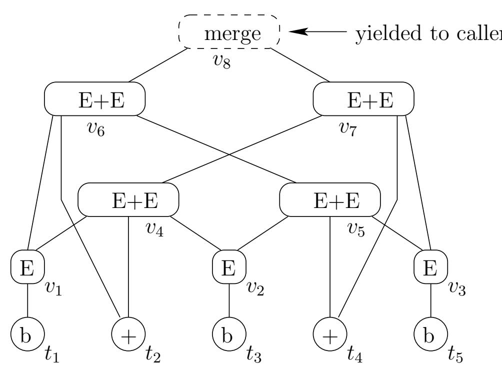


Fig. 3. A parse forest for the *EEb* grammar.

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, the value 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 3, v_1 has been used in the creation of v_4 and v_6.

Depending on what the actions do, there may be consequences to sharing that require user intervention. For example, if explicit memory management is being used, unrestricted sharing could lead to objects being deallocated multiple times; or, if the analysis is trying to count instances of some construct, sharing could cause some subtrees to be counted more than once.

To allow for a range of sharing 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 (`dup` immediately followed by `del`) 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 (this is the default behavior in Elkhound). 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. In any case, the design neither requires nor prohibits any particular memory management strategy.

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 3, 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 conventional GLR implementations, because reductions and merges are not always performed in a bottom-up order. As shown below, it is possible to yield a semantic value to a reduction action, only to then discover that the yielded value was ambiguous and needed to be merged. Because of this order violation, actions are severely constrained in what they can do, because they are given incomplete information.

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

$$\begin{aligned} S &\to A \\ A &\to d \ \mid \ B \\ B &\to d \end{aligned} \qquad (SAdB)$$

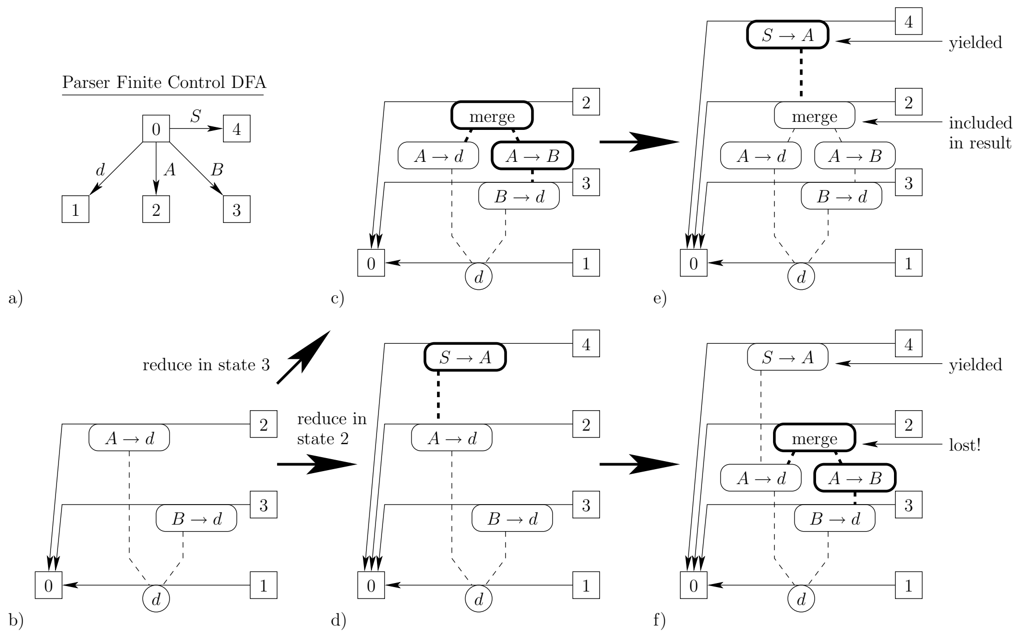


Fig. 4. There are two possible reduction sequences for the *SAdB* grammar, depending on reduction order. Square boxes are stack nodes, and rounded rectangles represents the result of running the user's reduction action code. Dotted lines show the flow of semantic values; they resemble parse tree edges.

For reference, Figure 4a shows the states of the finite control. In Figure 4b, the d has been shifted, creating the node with state 1, and the actions for $A \to d$ and $B \to d$ have been executed, creating nodes with states 2 and 3. The dotted lines reflect dependencies between semantic values. But from there the algorithm can proceed in two ways because there are two top nodes that can reduce.

First, Figures 4c and 4e show the sequence of GLR stack configurations when the node in state 3 reduces first. First (Figure 4c), the action for $A \to B$ runs. Then, because this reduction leads to another stack configuration with state 2 on top of (pointing to) state 0, the algorithm merges the two semantic values for nonterminal A as well as the stacks themselves. Finally, (Figure 4e) 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 $S \to A$. No information has been lost, and this is good.

On the other hand, Figures 4d and 4f show an alternative sequence, where state 2 reduces first. In that case, the action for $S \to A$ runs immediately (Figure 4d), and this is in fact the final result of the parse. Then (Figure 4f), 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, which is bad.

The problem stems from the fact that conventional GLR does the parsing work for each token using a worklist of stack nodes. Since each node is processed in its entirety, performing all reductions enabled at that node before the reductions for any other node, certain reductions are forced to occur together. Thus, it isn't always possible to pick an order for processing the stack nodes that will ensure bottom-order action execution.

To guarantee bottom-up actions, we must use a finer-grained worklist. Specifically, we will put *reduction* opportunities themselves in the list, instead of stack nodes. As each stack node is created, the algorithm computes the set of enabled actions. Shifts are postponed until the end of the phase (as in conventional GLR), and reductions are inserted into the worklist. The reduction worklist is maintained in a sorted order, according to this partial order:

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

In Figure 4, Rule 2 would force $A \rightarrow B$ to execute before $S \rightarrow A$ since $S \rightarrow^{+} A$. In general, reductions executed in this order are guaranteed to be bottom-up, assuming the grammar is acyclic.[5] The proof has been omitted, but appears in [9].

It should be noted that traditional GLR implementations cope with the action order problem by the way they construct their parse trees. For example, the algorithm described in [5] uses "Symbol" nodes to represent semantic values, which can be updated after being yielded since they are always incorporated by reference. Since the user's code is not run during parsing, it cannot observe that the semantic value is temporarily incomplete. We think that requiring users of Elkhound to write actions that are similarly tolerant would be too burdensome.

5 Case Study: A C++ Parser

To verify 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 post-parse disambiguator (a limited type checker), but not including support libraries. The grammar 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.

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

[6] Namespaces ([12] Section 7.3) and template partial specialization ([12] Section 14.5.4) are not currently implemented because they are not needed to parse Mozilla. We foresee no new difficulties implementing these features.

The C++ language definition [12] includes several provisions that make parsing the language with traditional tools 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.

The traditional solution for C, 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 categorizes the token as either a type name or a variable name.

In C++, the hack is considerably more difficult to implement, 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)
};
```

To make lexer hack work for C++, 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. Since the rules for name lookup and introduction are quite complex, a great deal of semantic infrastructure is entangled in the lexer feedback loop.

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 for classes 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.

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 ([12], Section 6.8) that if some syntax can be a declaration, then it is a declaration.

Using traditional tools that require LALR(1) grammars, and grammars for language fragments A and B, we would need to write a grammar for the language $A \backslash B$. This is at best difficult, and at worst impossible: the context-free languages (let alone LALR(1) languages) are not closed under subtraction.

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 (including with respect to names of types vs. names of variables) 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.

5.3 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 ([12], Section 14.2, par. 3). By recording grouping parentheses in the AST, we can select the correct parse with a simple pattern check once all possibilities have been parsed.

A correct implementation in an LALR(1) setting would again require recognizing a difference between languages. 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 violence 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 obscure 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.

5.4 Debugging Ambiguities

Building the C++ parser gave us a chance to explore one of the potential drawbacks of using the GLR algorithm, namely the risk that ambiguities would end up being more difficult to debug than conflicts. After all, at least conflicts are decidable. Would the lure of quick prototyping lead us into a thorn bush without a decidable boundary?

Fortunately, ambiguities have not been a problem. By reporting conflicts, Elkhound provides hints as to where ambiguities may lie, which is useful during initial grammar development. But as the grammar matures, we find ambiguities by parsing inputs, and understand them by printing the parse forest (an Elkhound option). This process is fundamentally different and more natural than debugging conflicts, because ambiguities and parse trees are concepts directly related to the *grammar*, whereas conflicts conceptually depend on the parsing *algorithm*. Consequently, debugging ambiguities fits easily among, and is in fact a very small portion of, other testing activities.

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 (itself written in C++), to test the claim that our algorithm enhancements in fact yield competitive performance. The experiments were performed on a 1GHz AMD Athlon running Linux. We report the median of five trials.

6.1 Parser Generators

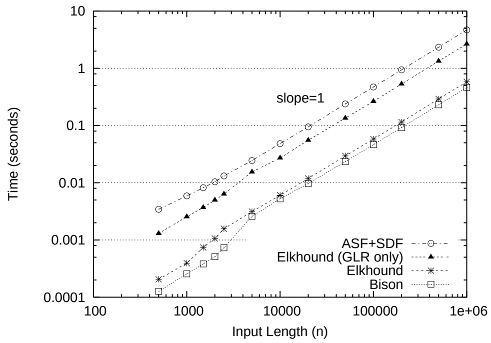
For comparison with an LALR(1) implementation, we compare Elkhound to Bison, version 1.28 [8]. Bison associates user action code with each reduction, and generates a parser written in C.

For comparison with an existing GLR implementation, we used the ASF+SDF Meta-Environment [7]. The Meta-Environment contains a variety of language-processing tools, among which is a scannerless GLR parser [13]. 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.

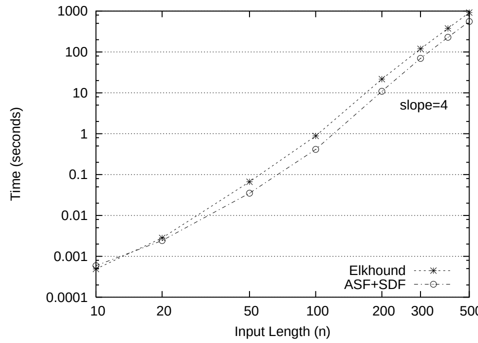
To measure the speed of Elkhound's ordinary LR parser, we measured its performance against Bison and ASF+SDF on the following LALR(1) grammar, also used in [14]:

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

All parsers were run with empty actions, i.e. as recognizers, and using trivial (one character is a token) lexers. As shown in Figure 5a, the Elkhound LR parser core is only about 10% slower than Bison, whereas ASF+SDF is a factor of ten slower than both. When Elkhound's hybrid mechanism (Section 3) 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.



a) *EFa* grammar, input is $a(+a)^n$.



b) *EEb* grammar, input is $b(+b)^n$.

Fig. 5. Parser performance (log-log scale).

To measure the full GLR parser, 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 $b(+b)^*$.

As shown in Figure 5b, Elkhound's performance is very similar to that of ASF+SDF. This suggests that the algorithm modifications previously presented, especially the use of a worklist of reductions instead of stack nodes (Section 4.2), does not substantially compromise GLR performance. Note that both require $\Theta(n^4)$ time; for grammars with such a high degree of ambiguity, GLR is often slower than the Earley algorithm [6].

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. We selected six of Mozilla's modules at random to measure.

Table 1 shows several measurements for each file. Parsing time is reported in milliseconds. "No-LR" is parse time when the LR parser is disabled, and (×) is the ratio of No-LR to Parse. The ratios 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 that the C++ parser cannot use the LR parser all the time; on the average, for our current grammar, it can use it only about 70% of the time. Further effort spent removing conflicts would presumably raise this percentage.

Table 1. C++ Parser Performance (times in ms).

		Elkhound			gcc-2.95.3	gcc-3.3.2
Preprocessed File Name	Lines	Parse	No-LR (×)	Disamb	Parse (×)	Parse (×)
nsUnicodeToTeXCMRt1.i	9537	16	36 (2.25)	50	60 (1.10)	80 (0.83)
nsAtomTable.i	19369	104	179 (1.72)	296	270 (1.48)	480 (0.83)
nsCLiveconnectFactory.i	24055	80	167 (2.09)	273	250 (1.41)	230 (1.53)
nsSOAPPropertyBag.i	26807	173	298 (2.30)	418	460 (1.28)	740 (0.80)
nsMsgServiceProvider.i	39215	209	378 (1.81)	545	560 (1.35)	1270 (0.59)
nsHTMLEditRules.i	49566	495	827 (1.67)	934	1140 (1.25)	2170 (0.66)

"Disamb" is the time for the post-parse disambiguator to run; the sum of the Parse and Disamb columns is the total time to parse and disambiguate. "gcc-2.95.3" and "gcc-3.3.2" are the time for two versions of gcc to parse the code as measured by its internal `parse_time` instrumentation, and (×) is the ratio of Elkhound total parse time to g++ parse time. Remarkably, the Elkhound C++ parser is typically only 30–40% slower than gcc-2, and usually faster than gcc-3.

7 Related Work

Performance In the worst case, the GLR algorithm takes $O(n^{p+1})$ time, where p is the length of the longest right-hand side. Converting the grammar to Chomsky normal form would thus make the bound $O(n^3)$ but would destroy conceptual structure. Kipps [15] suggests a different way to achieve $O(n^3)$ bounds, but with high constant-factor costs.

User-specified Actions ASF+SDF intends its users to write analyses using a sophisticated algebraic transformation (rewrite) engine. While the actions are not arbitrary, they are nonetheless powerful. The advantage is the availability of a number of high-level notations and tools; the disadvantage is the impedance mismatch with components outside the ASF+SDF framework. In particular, its internal ATerm [16] library prohibits destructive updates.

Bison allows user-specified actions, and recent versions of Bison include an extended parser that emulates GLR by copying the stack instead of building a graph, and executing user actions by traversing the parse forest afterwards. Copying the stack leads to exponential worst-case time, and tree traversal is expensive. It's intended for grammars with only minor deviations from LALR(1).

Disambiguation Many systems support some form of declarative disambiguation. Several static conflict resolution schemes such as precedence and associativity [17] or follow restrictions [18] are implemented in Elkhound. Since it uses a conventional LR finite control, it is straightforward to support such schemes.

Dynamic, parse-time disambiguation in Elkhound is supported through the `keep` and `merge` functions. ASF+SDF has related `reject` and `prefer` directives. `reject` does declaratively what `keep` does imperatively, namely to recognize a language difference such as $A \backslash B$. When B is already described as a nonterminal in the grammar, `reject` is more convenient; when B is ad-hoc or context-dependent, the flexibility of `keep` is handy. Similarly, `prefer` can be simulated imperatively by `merge`, but `merge` can also choose to retain the ambiguity.

Wagner and Graham [19] argue for post-parse semantic disambiguation, in an incremental GLR parsing setting. We follow their lead with a batch parser.

Other Algorithms The first parsing algorithm for the entire class of context free grammars was the Earley dynamic programming algorithm [6], with running time $\Omega(n^2)$ and $O(n^3)$. A number of variations and refinements have been proposed [20, 21, 14, 22, 23], but none has yet emerged as a practical algorithm for parsing programming languages.

By adding backtracking to a deterministic algorithm, one can achieve the effect of unbounded lookahead. Conventional systems that do this include BtYacc and ANTLR; one can also use higher-order combinators in a functional language to do the same [24]. However, since they do not yield all parses for an ambiguous grammar, many of the techniques presented in Section 5 would not be possible.

8 Conclusion

This paper presents two key enhancements to the GLR algorithm. First, the GLR/LR hybrid substantially improves performance for deterministic grammar fragments. Second, the use of a reduction worklist instead of a node worklist enables user-defined actions to be executed in bottom-up order.

We then demonsrate the benefits of each improvement by the process of constructing a parser for C++. User-written actions, especially during ambiguity merging, are capable of effectively disambiguating troublesome constructs, and the resulting parser comes close to matching the speed of a production compiler.

We believe GLR parsing is a valuable tool for language research, and towards that end Elkhound and its C++ parser have been released under an open-source (BSD) license, available at
`http://www.cs.berkeley.edu/~smcpeak/elkhound` .

References

1. Johnson, S.C.: YACC: Yet another compiler compiler. In: UNIX Programmer's Manual (7th edn). Volume 2B. (1979)
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1986)
3. Lang, B.: Deterministic techniques for efficient non-deterministic parsers. In Loeckx, J., ed.: Automata, Languages and Programming. Volume 14 of Lecture Notes in Computer Science. Springer (1974) 255–269
4. Tomita, M.: Efficient Parsing for Natural Language. Int. Series in Engineering and Computer Science. Kluwer (1985)
5. Rekers, J.: Parser Generation for Interactive Environments. PhD thesis, University of Amsterdam, Amsterdam, The Netherlands (1992)

6. Earley, J.: An efficient context-free parsing algorithm. Communications of the ACM **13** (1970) 94–102
7. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The syntax definition formalism SDF - reference manual. SIGPLAN Notices **24** (1989) 43–75
8. Donnelly, C., Stallman, R.M.: Bison: the YACC-compatible Parser Generator, Bison Version 1.28. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139 (1999)
9. McPeak, S.: Elkhound: A fast, efficient GLR parser generator. Technical Report CSD-02-1214, University of California, Berkeley (2002)
10. Knuth, D.E.: On the translation of languages from left to right. Information and Control **8** (1965) 607–639
11. Nozohoor-Farshi, R.: GLR parsing for ϵ-grammars. In Tomita, M., ed.: Generalized LR Parsing. Kluwer (1991) 61–75
12. International Organization for Standardization: ISO/IEC 14882:1998: Programming languages — C++. International Organization for Standardization, Geneva, Switzerland (1998)
13. Visser, E.: Scannerless generalized-LR parsing. Technical Report P9707, University of Amsterdam (1997)
14. Alonso, M.A., Cabrero, D., Vilares, M.: Construction of efficient generalized LR parsers. In: WIA: International Workshop on Implementing Automata, LNCS, Springer-Verlag (1997)
15. Kipps, J.R.: GLR parsing in time $O(n^3)$. In Tomita, M., ed.: Generalized LR Parsing. Kluwer (1991) 43–60
16. van den Brand, M., de Jong, H.A., Klint, P., Olivier, P.A.: Efficient annotated terms. Software Practice and Experience **30** (2000) 259–291
17. Earley, J.: Ambiguity and precedence in syntax description. Acta Informatica **4** (1975) 183–192
18. van den Brand, M., Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation filters for scannerless generalized LR parsers. In: Compiler Construction. (2002) 143–158
19. Wagner, T.A., Graham, S.L.: Incremental analysis of real programming languages. In: ACM Programming Language Design and Implementation (PLDI). (1997) 31–43
20. Graham, S.L., Harrison, M.A., Ruzzo, W.L.: An improved context-free recognizer. ACM Transactions on Programming Languages and Systems (TOPLAS) **2** (1980) 415–462
21. McLean, P., Horspool, R.N.: A faster Earley parser. In: Compiler Construction. (1996) 281–293
22. Schröer, F.W.: The ACCENT compiler compiler, introduction and reference. Report 101, German National Research Center for Information Technology (2000)
23. Aycock, J., Horspool, R.N., Janoušek, J., Melichar, B.: Even faster generalized LR parsing. Acta Informatica **37** (2001) 633–651
24. Hutton, G.: Higher-order functions for parsing. Journal of Functional Programming **2** (1992) 323–343