Incremental Analysis of Real Programming Languages[*]

Tim A. Wagner and Susan L. Graham
University of California, Berkeley

Abstract

A major research goal for compilers and environments is the automatic derivation of tools from formal specifications. However, the formal model of the language is often inadequate; in particular, LR(k) grammars are unable to describe the natural syntax of many languages, such as C^{++} and Fortran, which are inherently non-deterministic. Designers of batch compilers work around such limitations by combining generated components with ad hoc techniques (for instance, performing partial type and scope analysis in tandem with parsing). Unfortunately, the complexity of *incremental* systems precludes the use of batch solutions. The inability to generate incremental tools for important languages inhibits the widespread use of language-rich interactive environments.

We address this problem by extending the language model itself, introducing a program representation based on *parse dags* that is suitable for both batch and incremental analysis. Ambiguities unresolved by one stage are retained in this representation until further stages can complete the analysis, even if the resolution depends on further actions by the user. Representing ambiguity explicitly increases the number and variety of languages that can be analyzed incrementally using existing methods.

To create this representation, we have developed an efficient incremental parser for general context-free grammars. Our algorithm combines Tomita's generalized LR parser with reuse of entire subtrees via state-matching. Disambiguation can occur statically, during or after parsing, or during semantic analysis (using existing incremental techniques); program errors that preclude disambiguation retain multiple interpretations indefinitely. Our representation and analyses gain efficiency by exploiting the local nature of ambiguities: for the SPEC95 C programs, the explicit representation of ambiguity requires only 0.5% additional space and less than 1% additional time during reconstruction.

[*]This research has been sponsored in part by the Defense Advanced Research Projects Agency (DARPA) under Grant MDA972-92-J-1028. The content of this paper does not necessarily reflect the position or policy of the U. S. Government.
Authors' addresses: Tim A. Wagner, 573 Soda Hall and Susan L. Graham, 771 Soda Hall; Department of EECS, Computer Science Division, University of California, Berkeley, CA 94720-1776.
email: twagner@cs.berkeley.edu, graham@cs.berkeley.edu
URL: http://http.cs.berkeley.edu/~twagner,
http://http.cs.berkeley.edu/~graham.

1 Introduction

Generating compiler and environment components from declarative descriptions has a number of well-known advantages over hand-coded approaches, especially when the result is intended for an incremental setting. However, existing formal methods use limited—and unrealistic—language models. In particular, *ambiguity*, in both syntactic and semantic forms, is outside the narrow constraints of LR(1) parsing (the conventional method for syntax analysis) and is not addressed by attribute grammars (the most common form of formal semantic analysis).

Batch systems cope with such language 'idiosyncrasies' by remaining open; ad hoc code is coupled with generated components to overcome limitations in the language model. Those solutions succeed because the language document is static and the analysis order is fixed. (For example, it can be assumed that necessary symbol table information is available when needed.) The greater complexity of incremental algorithms precludes simple ad hoc solutions, due to the need to support incomplete documents and partial analyses that depend on the order in which the user modifies the program. The result is a collection of standard representations and algorithms unable to directly model the analysis of C, C^{++}, Fortran, Haskell, Oberon, and many other languages. Thus many potential applications—compilers, environments, language-based tools—forgo incrementality in favor of slower, less informative batch technologies.

Rather than lament the design of these languages, we address the underlying issue by extending the language model, producing a framework that allows existing formalisms to apply to a wider variety of languages. Our solution utilizes a new intermediate representation (IR) for the early portions of the (possibly incremental) compilation pipeline: the *abstract parse dag* allows multiple interpretations to be represented directly and efficiently. The familiar pass-oriented compiler organization is supported, even in incremental settings, by allowing ambiguities to be resolved at different stages of the analysis. Semantic filters address the 'feedback' problem (syntactic structure dependent upon semantic information) arising in C and Fortran. Parsing filters [11] address such problems as the declaration/expression ambi-

```
int foo () {
  int i;
  int j;
  a (b); ← ambiguous—could be
  c (d); ← decls or stmts.
  i = 1;
  j = 2;
}
```

Figure 1: A simple example of ambiguity in C and C^{++}. In this case, type information is necessary for disambiguation: the middle two lines can be either declarations or function calls, depending on how a and c have been declared previously in enclosing scopes.

guity in C^{++} [3] and the ‘off-side’ rule in Haskell [7]. We describe mechanisms for applying both types of resolution using existing formal techniques, such as attribute grammars, while also permitting ad hoc resolution. Pre-compiled filters such as precedence and associativity declarations in yacc [1] are supported in a uniform fashion. In the presence of missing or malformed program text, multiple interpretations may be retained indefinitely as a direct expression of the possibilities.

We have developed a novel algorithm for incremental, non-deterministic parsing to (re)construct this IR. The parser accepts all context-free grammars: generalized LR parsing [20, 22] is used to support non-determinism and ambiguity, eliminating restrictions on the parsing grammar and the attendant need for abstraction services. Shifting of entire subtrees via state-matching [8] provides efficient incremental behavior, and explicit node retention [25] minimizes the work of subsequent analysis passes. (Together they also ensure the preservation of user context and program annotations.) Lookahead information is dynamically tracked and encoded in parsing states stored in the nodes, eliminating the space overhead of previous approaches that require persistent maintenance of the entire graph-structured parse stack [4].

As an example of an inherent context-free syntax ambiguity addressed by this representation, consider the syntax of C. Figure 1 illustrates a case where the interpretation of several lines is context-sensitive, i.e., ‘static semantic’ analysis is needed to resolve the ambiguity.[1] A similar problem arises in C^{++}, Fortran, Oberon, and other languages. This problem arises whenever the natural context-free syntax depends on non-local type information [28].

Ambiguity is discovered during analysis of the context-free syntax, leaving multiple alternatives encoded in the parse dag. Early stages of semantic analysis resolve typedef declarations; binding information for type names is then used to complete the resolution of the program’s syntax. (In the case of a correct program, the parse dag will become a conventional abstract parse tree.) Semantic analysis then continues, using the resolved structure. This approach preserves the familiar compilation pipeline model, and allows existing formal methods to be applied to C and other ‘ill-designed’ languages to produce either batch or incremental environments.

Encoding alternatives for later resolution is useful in a number of stages in the compilation pipeline. Lexical decisions are often deferred until parsing or semantic analysis by having the lexer recognize only equivalence classes of tokens. Visser [24] makes this integration explicit for a batch system by using a single GLR parser for both lexical *and* context-free analysis. This approach can be made incremental using the techniques we describe. Code generation also benefits from retaining multiple representations until additional information has been gathered. Giegerich [5] applies context-sharing in this domain to intersperse code selection and register allocation.

We have measured the space costs of our representation and the time overhead to rebuild it incrementally using a benchmark suite that includes both C^{++} programs and the C programs in SPEC95. Both measurements indicate that the significant increase in the flexibility of the language model comes at virtually no cost. The efficiency results from exploiting an inherent property of programming (and natural) languages: ambiguity is both constrained (the number of interpretations is small) and localized (the length of an ambiguous construct is limited).

The remainder of this paper is organized as follows. In Section 2 we describe the basic form of the program representation, concentrating on the handling of alternative interpretations. Section 2 also summarizes empirical studies demonstrating the highly localized nature of ambiguity in programs and the minimal space overhead achievable through sharing. In Section 3 we consider in detail the construction of our program representation using an incremental, non-deterministic parser. We introduce a performance model and analyze the asymptotic behavior of the parser to demonstrate the efficiency of incremental updates. We conclude this section with a return to the issue of sharing in the abstract parse dag, demonstrating optimality and correctness properties unique to our method. Mechanisms for disambiguation at various points in the analysis phase—particularly semantic disambiguation involving type information—are presented in Section 4. Implementation details and empirical comparisons between deterministic parsing/parse trees and non-deterministic parsing/abstract parse dags are given in Section 5. A discussion of future work and our conclusions end the paper. The incremental GLR parsing algorithm is provided in Appendix A. A trace of the parser actions on a small C^{++} example is given in Appendix B.

2 Representing Ambiguity

A phase-oriented incremental system can succeed only if the intermediate representation explicitly represents unresolved ambiguities. The *abstract*

[1] Batch systems typically handle this problem by having the lexer query the symbol table in order to separate identifiers into two distinct categories. Attribute-influenced parsing [10, 21] is a combination of LR parsing and a restricted class of attribute grammars that addresses the same problem in a formal way. Neither of these solutions can be applied to an incremental setting where non-trivial subtrees appear in the parser’s input stream.

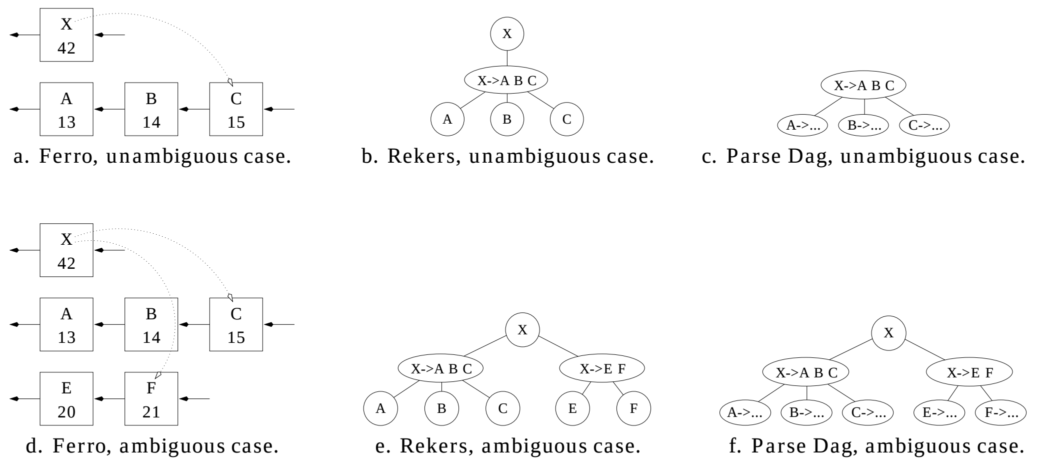

X
42
A
13
B
14
C
15
a. Ferro, unambiguous case.
X
X->A B C
A
B
C
b. Rekers, unambiguous case.
X->A B C
A->...
B->...
C->...
c. Parse Dag, unambiguous case.
X
42
A
13
B
14
C
15
E
20
F
21
d. Ferro, ambiguous case.
X
X->A B C
X->E F
A
B
C
E
F
e. Rekers, ambiguous case.
X
X->A B C
X->E F
A->...
B->...
C->...
E->...
F->...
f. Parse Dag, ambiguous case.

Figure 2: Comparison of the abstract parse dag to other proposed representations. The grammar productions illustrated are X→ABC | EF. Ferro and Dion's approach (a) makes the GSS itself persistent; this requires semantic attributes associated with a production (right-hand side) to be attached to a constellation of nodes rather than an individual object. Rekers' representation (b) is more like a classic parse tree but separates the symbol (phylum, left-hand side) and rule (production, right-hand side) into separate nodes. This imposes significant overhead, since the vast majority of the program is deterministic. Our approach represents the deterministic portions of the tree in the conventional manner (c), using Rekers-style splitting only where multiple representations actually exist (f). (Not shown are the additional state collections required by the Ferro and Dion approach or the problems with under- and over-sharing of epsilon productions eliminated in the abstract parse dag.)

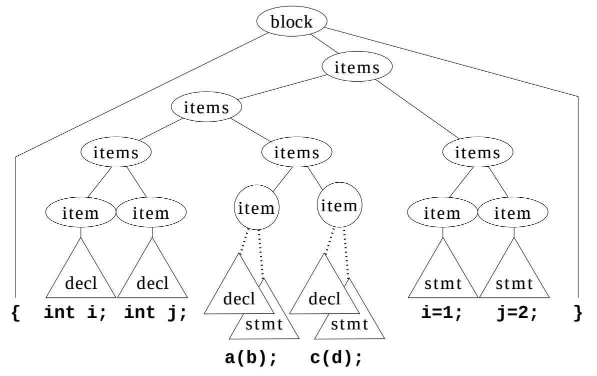


Figure 3: Representation of ambiguous structure in the abstract parse dag. This is the result of parsing the example in Figure 1 as a C^{++} program. Most nodes represent both productions and symbols. *Choice points*, shown as circles, represent only symbols; their children comprise the alternative interpretations. In this case the shared subtrees are trivial—they are the terminal symbols in the ambiguous region. The structure shown represents a simplification of the complete grammar.

parse dag is similar to a parse tree except that a region may have multiple interpretations. This section describes the representation itself; subsequent sections describe its construction, via non-deterministic parsing, and the resolution of ambiguities expressed through this IR.

In the presence of ambiguity, many parse trees potentially represent the program. To avoid exponential blowup, this entire forest is collapsed into a single, compact data structure. *Subtree sharing* merges isomorphic regions from different trees, and requires no special changes—each instance of a production is represented by a single node, just as in a parse tree. Merging *contexts*,[2] however, requires a new type of node to indicate the choices. A *symbol node* represents a phylum (left-hand side) instead of an entire production; its children represent the possible interpretations of their common yield. In the case of a correct program, later stages of analysis will disambiguate the program by selecting exactly one child of each symbol node. Figure 2 illustrates the distinction between symbol and production nodes and compares our representation to other proposals. Figure 3 shows the abstract parse dag corresponding to the example in the introduction.

If the number of alternate interpretations at a single point is large, the children of a symbol node can be represented as a balanced binary tree to ensure the performance characteristics described in Section 3.4. In practice, however, the number of alternatives is effectively bounded and a simple list provides sufficiently fast access.

In a typical batch compiler, a grammar from a restricted grammar class is used to produce a parser for the concrete syntax. A separate (often implicit) grammar defines the abstract syntax representation

[2]Sometimes referred to as 'packing' in natural language analysis.

Program	Lines	Lang	%ov
compress	1934	C	0.21
gcc	205093	C	0.10
go	29246	C	0.00
ijpeg	31211	C	0.02
m88ksim	19915	C	0.02
perl	26871	C	0.01
vortex	67202	C	0.00
xlisp	7597	C	0.02
emacs 19.3	159921	C	0.47
ensemble	294204	C++	0.26
idl 1.3	29715	C++	0.10
ghostscript 3.33	128368	C	0.52
tcl 7.3	26738	C	0.31

Table 1: Programs used in this study. The first eight are from SPEC95. `idl` is the SunSoft IDL front end and `ensemble` is our prototype software development environment.

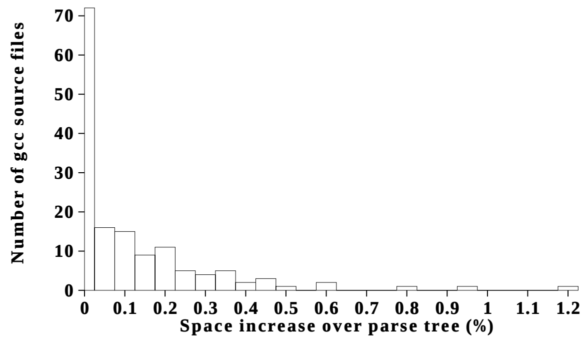


Figure 4: Distribution of ambiguities by source file in `gcc`. This histogram groups the source files of `gcc` according to the amount of syntactic ambiguity they possess. The syntax of C++ was used to determine these counts; the percentages would be lower using a C grammar, due to the more restrictive statement syntax of that language. All ambiguities are semantically resolved (the 'typedef problem'). They consist of two interpretations each, and share only terminal symbols.

of the parsed program after artifacts of the concrete parse have been removed. GLR parsing enables a *single* grammar to formally define both the representation and the mechanism that builds it: support for multiple syntactic interpretations and nondeterministic parsing permit arbitrary CFGs to be used in describing the language. This generality allows the grammar to serve as a pure definition of the resulting structure, rather than requiring it to conform to the restrictions of some particular parsing class.[3] Since our parse dag representation inherits this benefit of GLR parsing, we refer to it as 'abstract'. (We will sometimes omit this modifier.)

The abstract parse dag differs from the ordinary shared forest discovered by a GLR parser: Instances of productions are always represented by individual nodes, and sharing of both subtrees and contexts is optimal. We return to issues of sharing in Section 3.5 after explaining incremental GLR parsing.

2.1 Space Overhead for Ambiguity

Cognitive studies suggest that localization of ambiguity is an inherent property of natural languages, a constraint imposed by limitations on short-term memory [15, 17]. Our studies find an identical result for programming languages.[4] Since an abstract parse dag exploits localization of ambiguity through the sharing of subtrees and contexts, the increase in space required relative to a fully disambiguated parse tree provides an ideal measure of the amount of ambiguity (as well as the space overhead of adopting our IR). For the suite of C and C++ programs in Table 1, we measured the increased space consumption required to represent the multiple interpretations of each syntactically ambiguous construct. The increase is relative to the parse tree produced by a batch compiler (using semantic feedback to the lexer and with the corresponding ambiguity in the grammar resolved through different identifier namespaces). The average increase for each program in the suite is shown in the final column of Table 1. Figure 4 shows the ambiguity distribution by source file for `gcc`.

3 Constructing the Abstract Parse Dag

We now consider the construction of the abstract parse dag via incremental, non-deterministic parsing. We first review batch GLR parsing and incremental parsing, which will jointly form the basis for the incremental GLR (IGLR) parser. We introduce a performance model to analyze the asymptotic behavior of the parser, and conclude the section by proving that sharing in the abstract parse dag is both optimal and correct. The algorithm itself appears in Appendix A.

3.1 Generalized LR Parsing

Batch GLR parsing [18, 20, 22] is a technique for parsing arbitrary context-free grammars that utilizes conventional LR table construction methods. Unlike deterministic parsers, however, a GLR parser permits these tables to contain conflicts: when a state transition is multiply-defined, the GLR parser simply forks multiple parsers to follow each possibility. In the case of a deterministic parse requiring additional lookahead, all but one of these parsers will eventually terminate by encountering a syntax error. In the case of true ambiguity, multiple valid representations will be discovered. In both cases, the *graph-structured parse stack* (GSS) represents the combined parse stacks compactly. This sharing is made possible by having the GLR parse proceed

[3] Even with GLR parsing, some erasing of concrete elements unnecessary for the abstract structure, such as parentheses, is often done.

[4] This property was indirectly measured by Tomita [22] and Rekers [20], who compared the speed of a batch GLR parser to Earley's algorithm [2] on natural and programming language grammars, respectively. Both authors concluded that grammars are 'close' to LR(1) in practice, and therefore GLR parsing exhibits linear behavior despite its exponential worst-case asymptotic result.

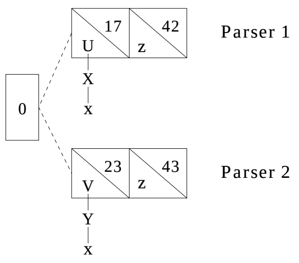


Figure 5: Illustration of non-determinism in a GLR parser. When the grammar is ambiguous or requires lookahead greater than that of the table construction method (typically a single terminal), a GLR parser will *split* into two or more parsers. Here two parsers are being used in a region requiring two terminals of lookahead with an LR(1) table. (The grammar appears in Figure 7.) In this case the parse is non-deterministic but unambiguous: when sufficient lookahead has been scanned dynamically, the GLR automaton will collapse back to a single parser. In cases of true ambiguity, multiple interpretations are preserved in the resulting abstract parse dag.

breadth-first: each terminal symbol is shifted simultaneously by all active parsers in the collection. Figure 5 illustrates a GLR parser processing the non-LR(1) grammar of Figure 7.

As demonstrated in batch environments, GLR parsing simplifies the specification of programming languages by removing restrictions on the parsing grammar and eliminating the need for a separate abstraction mechanism. The ability to use additional lookahead allows a more natural expression of syntax and enables the description of truly ambiguous languages.

3.2 Incremental Parsing

In an incremental parser, the input stream consists of both terminal *and nonterminal* symbols; the nonterminals label the roots of the unmodified subtrees from the previous version of the parse tree. Two distinct approaches can be taken: *sentential-form parsing*, where the grammar is the basis for incrementality, and *state-matching*, where the configuration of the pushdown automaton is recorded in the tree and used to skip steps in subsequent analyses. For LALR or LR grammars with all conflicts resolved at parse table construction time, sentential-form parsing is the better implementation method, since it requires less time and space than a state-matching algorithm [25]. However, sentential-form parsing cannot be used as the basis for a *non-deterministic* incremental parser with conventional table construction: the stronger test of state-matching is needed to expose the possibility of non-deterministic splitting when shifting an otherwise valid subtree.

In a state-matching implementation [8, 14], each node representing a nonterminal symbol contains a record of the configuration of the pushdown automaton (the 'parse state') when the node was shifted onto the stack. A subtree can be reused when both its left and right context are unchanged: in an LR(1) parser, reuse is determined by an equality test between the current parse state and the state recorded

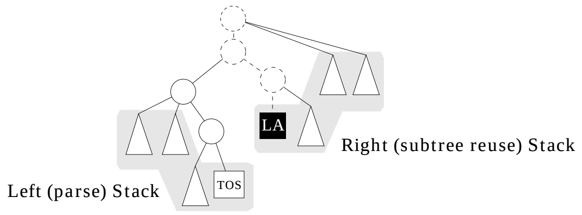


Figure 6: Illustration of deterministic incremental parsing. Here a change to a token has resulted in a split of the parse tree from the root to the shaded lookahead node denoting the modified terminal symbol. (Nodes on the split path are shown dashed.) The shaded region to the left is the parse stack, which is instantiated as a separate data structure since it contains a mixture of old and new subtrees. The shaded region to the right is the subtree reuse stack, which provides the potentially reusable subtrees of the parser's input stream. This stack is not explicitly materialized—its contents are derived by a traversal of the parse tree as it existed immediately prior to reparsing.

in the node, together with a check to ensure that the same terminal symbol follows the subtree as in the previous analysis. If the shift of a subtree is invalid, it is decomposed into its constituent subtrees, which are pushed back onto the input stack. This process continues until the lookahead symbol is a terminal or shifting can resume.

The user may apply any number of changes before requesting a reparse. Both textual and structural editing are permitted; the structure of the parse dag and the contents of its terminal symbols (tokens) reflects all modifications applied since the previous parse. Once the parser is invoked, it 'splits' the parse dag at each modification point (interior nodes with structural changes or terminal nodes with textual changes). The input stream to the parser consists of both new material, in the form of tokens provided by an incremental lexer, and reused subtrees; the latter are conceptually on a stack, but are actually produced by a directed traversal over the version of the tree as it existed immediately prior to the start of reparsing [26]. An explicit stack is used to maintain the new version of the tree while it is being built. Figure 6 illustrates a common case, where a changed token has resulted in a split from the root to the changed terminal symbol.

Shifting a subtree takes $O(1)$ time when state-matching succeeds. However, *reductions* require extra time since a terminal symbol is needed to index the parse table. (Alternatively, the leftmost terminal descendant can be recorded in every node, costing space.) Often this overhead can be eliminated entirely by precomputing nonterminal reductions: we can perform reductions with a nonterminal N if all reduction actions in state s are identical for every terminal in FIRST(N), provided that N does not generate ϵ. In the remaining cases, the lookahead's structure must be traversed to locate the next terminal.

3.3 Incremental GLR Parsing

We now turn to the construction of an *incremental* GLR (IGLR) parser that can parse an arbitrary CFG non-deterministically, while simultaneously accepting non-trivial subtrees in its input stream. The abstract parse dag is (re)created during parsing; Section 3.5 explores this process in more detail. Appendix B contains a sample trace of the IGLR parser's actions using our running example and a simplified C++ grammar.

The IGLR parser combines subtree reuse in deterministic regions with GLR methods in areas requiring non-deterministic parsing. This aggregation of the two algorithms is complicated by the unconstrained lookahead of non-deterministic parsing: even though such regions are limited in practice, locating the *boundary* of such a region is necessary in order to reuse unchanged subtrees.

As in previous GLR algorithms, we employ a graph-structured parse stack (GSS) to permit non-deterministic parsing. During parsing, deterministic behavior is assumed to be the common case. (Sections 2.1 and 5 validate this assumption through empirical measurements.) As with a deterministic state-matching parser, each node of the parse dag requires an additional word of storage to record the parse state in which it was constructed. LALR(1) tables are used to drive the parser: not only are they significantly smaller than LR(1) tables, but they also yield faster parsing speeds in non-deterministic regions [13] and improved incremental reuse in deterministic regions (due to the merging of states with like cores).[5]

Left context checks involve the same integer comparison used by a deterministic state-matching incremental parser. When elements of the parse are non-deterministic, however, the right context check is more complicated than its deterministic counterpart, which simply verifies that the terminal symbol following a potentially reusable subtree is unchanged. For general context-free parsing, there is no fixed bound on right context; an incremental GLR parser cannot assume that the amount of lookahead encoded in the parse table (usually one) is sufficient to determine when a reduction's right context is unchanged.

Instead, the incremental GLR parser must track lookahead use *dynamically*; this information is recorded in the nodes of the abstract parse dag, where it is used to influence future parses. The use of extended right context can be encoded in the same field normally used to record the parse state. *All* non-deterministic states are represented as an equivalence class with a unique state value. When any node possessing this state value occurs as the lookahead symbol in subsequent analyses, the matching test will fail and the parser will decompose the lookahead into its constituent subtrees.

Additional (dynamic) lookahead is required only when several parsers are simultaneously active. The IGLR parsing algorithm tracks this condition with a boolean flag. After shifting the lookahead, the flag is set to true if there are multiple active parsers. The flag is also set to true when a parse table interrogation returns multiple actions. During a reduction, the state value recorded in the newly-created dag node is the state of the single active parser, if the flag is false, and the value representing all non-deterministic states (and thus the use of additional lookahead), if the flag is true. Figure 7 shows a simple case where dynamic lookahead is used by our IGLR parser to analyze an LR(2) grammar using LR(1) tables.

[5] In the case where the grammar is LR but not LALR, the IGLR parser will try all the conflicting reductions, resolving the uncertainty when it shifts the following terminal symbol.

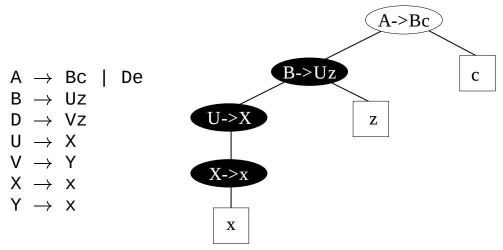


Figure 7: Tracking lookahead information dynamically. This example illustrates a grammar that requires two tokens of lookahead. A GLR parser based on a single-lookahead table will require non-determinism to parse the sentence xzc. Since the grammar is unambiguous, a unique parse tree results after c is read. The unsuccessful parser is discarded. Black ellipses indicate nodes for which increased lookahead must be recorded during parsing; note that they coincide with reductions performed while more than one parser was active. Nonterminals representing reductions in a deterministic state (A→Bc) require only the implicit (one token) lookahead; they are marked with the (singleton) parser's state when they are shifted onto its parse stack.

When both the previous state (preserved in the root node of the lookahead subtree) and the current state are deterministic, parsing proceeds as in Section 3.2. Shifted subtrees may contain non-deterministic areas as long as they are not exposed. Subtrees containing modifications (textual and/or structural edits) are decomposed to expose each change site. Subtrees from non-deterministic regions are similarly broken down, triggered by a failure of the normal state matching test. If a conflict is encountered, the parser splits just as in batch GLR parsing, and subtrees in the input stream are fully decomposed until a deterministic state is re-established (see the `shifter` routine in Appendix A).

Shifting an unmodified, non-trivial subtree condenses a sequence of transitions by the corresponding batch GLR parser. The portion of the abstract parse dag reused when the incremental algorithm shifts a non-trivial subtree reflect any splitting or merging that would occur in the GSS of the batch algorithm as it parsed the subtree's terminal yield. The correctness of skipping the intermediate steps is guaranteed in deterministic states by the usual incremental context checks, and in non-deterministic states (which are treated as an equivalence class) by the restriction to terminal lookaheads. The correctness of incremental GLR parsing can then be established by an induction over the input stream.

Our approach differs significantly from the non-deterministic PDA simulator of Ferro and Dion [4], which uses the GSS itself as the persistent representation of the program. Their representation requires more space than our parse dag, in part because unsuccessful parses (used to overcome lookahead limitations) must be retained for the sake of future state comparisons. (In Figure 5, the portion of the GSS constructed by Parser 2 must be kept, even though it represents an unsuccessful search.) Their algorithm also makes state comparisons and semantic attribution more expensive, since both must refer to a *collection* of nodes.

As with deterministic parsing, IGLR parsing can be extended to retain existing program structure through *node reuse* [14, 19, 25]. Both ambiguous and unambiguous reuse models are valid for abstract parse dags, and both bottom-up and top-down reuse mechanisms can be applied. (For on-the-fly bottom-up reuse, we advocate retaining a single, shared list of reused nodes; maintaining separate lists when multiple parsers are active imposes a performance and complexity cost for minimal gain in the number of reused nodes.)

3.4 Asymptotic Analysis

The IGLR parsing algorithm works for any context-free grammar and, like GLR parsing, is exponential in the worst-case [9] but linear on actual programming language grammars. To ensure incremental performance that improves on batch parsing, we need to impose some additional restrictions on both the grammar and the representation of the abstract parse dag.

Incremental behavior requires that the abstract parse dag support logarithmic access time to each node. This is *not* the normal case: repetitive structure, such as sequences of statements or lists of declarations, is typically expressed in grammars and represented in trees in a left- or right-recursive manner. These parse 'trees' are thus really linked lists, with the concomitant performance implication: any incremental algorithms degenerate to at best linear behavior, and thus represent no asymptotic advantage over their batch counterparts.

There are two types of operators in grammars that create recursive structure: those that might have semantic significance, such as arithmetic operators, and those that are truly associative, such as the (possibly implicit) sequencing operators that separate statements. The former do not represent true performance problems because they are naturally limited; for instance, we can assume that the size of an expression in C is bounded by a constant in practice. The latter type are problematic, since they are usually quite lengthy in non-trivial programs.

To address this problem, we represent associative operators in the abstract parse dag as *balanced binary trees*. An obvious way to indicate the freedom to choose a balanced representation for associative sequences is to describe the syntax of the language using an extended context-free (regular right part) grammar [12]. We thus use the grammar both to specify the syntax of the language and to describe declaratively the representation of the resulting abstract parse dag. Productions in the grammar correspond directly to nodes in the tree, while regular expressions denoting sequences have an internal representation chosen by the system—one that is guaranteed to maintain logarithmic performance. For purposes of analysis, we assume that any unbounded sequences are expressed in this fashion in the grammar. (Note that changes to the grammar are *required*—the parser generator cannot infer that a given sequence is associative.)

In addition, we need to assume that no non-deterministic region spans a lengthy sequence, since this would naturally require the entire sequence to be reconstructed whenever any part of it was changed. (Note that the *elements* of the sequence can be parsed non-deterministically or even be ambiguous, as is the case with C++.) Similarly, the interpretation of a sequence's yield cannot have more than a bounded dependence on its surrounding context, so that changes to adjacent material will not induce a complete reconstruction of the sequence.

Given this assumption regarding the form of the grammar and the representation of the abstract parse dag, we can analyze the time performance of the IGLR parser. In the typical case where the left and right context of a subtree are unchanged, a state-matching algorithm will shift that subtree in $O(1)$ time. In the event the context *has* changed, a valid subtree containing M nodes can be shifted in $O(\lg M)$ steps by reconstructing its leading or trailing edge. Reductions and the deterministic right context check are often accomplished in $O(1)$ time using the following subtree; in the worst case the following terminal symbol is located in $O(\lg M)$ steps. Locally non-deterministic regions are reconstructed in their entirety, but our assumption that the size of such regions is effectively bounded (Section 2.1) implies a constant bound on the time to parse them. The result is a typical parsing time of $O(t + s \lg N)$, for t new terminal symbols and s modification sites in a tree with N nodes, and $O(t + s(\lg N)^2)$ time in the worst case. (Empirical results are discussed in Section 5.)

3.5 Correct and Optimal Sharing

Our approach treats the GSS as a transient data structure of the parser, using it to construct the abstract parse dag in the same way deterministic parsers construct a concrete parse tree with the help of a parse stack. However, the connection is more complex than in the deterministic case. In this section we discuss the removal of parsing artifacts from the shared parse forest discovered by GLR methods to produce the representation described in Section 2.

The parse forest produced by GLR parsing results in both over- and under-sharing, complicating (in some cases precluding) the application of existing methods for semantic attribution and similar tools. GLR parsing as originally defined [22] results in *under*-sharing in the shared parse forest when isomorphic subtrees with the same yield are created in different states (i.e., by different parsers) due to left or right contextual restrictions.[6] Rekers corrects

[6]This is the same effect that causes incremental deterministic parsers based on state-matching to fail to reuse subtrees as aggressively as sentential-form parsers [25].

under-sharing in his batch GLR parser by merging nodes that have identical yields [20]. Merging is performed separately for both symbol and 'rule' (production) nodes. The same approach can be applied in our algorithm, since non-deterministic regions are reconstructed atomically.

A different problem exhibited by GLR algorithms is *over*-sharing. A GLR parser does not distinguish non-determinism to acquire additional lookahead information from its use in parsing ambiguous phrases. In most cases, non-determinism for dynamic lookahead results in deterministic (and unshared) structure in the parse tree, since unsuccessful parses eventually terminate. In the GSS, however, sharing needed to handle certain types of grammars with ϵ-productions results in sharing in the parse tree *even for unambiguous grammars* [18]. We consider this a flaw; among other problems, it prohibits semantic attributes or annotations from being uniquely assigned to productions with a null yield, since separate instances may not exist in the parse tree. (Rekers' algorithm exacerbates this problem by merging additional null-yield subtrees, violating left-to-right ordering.) We correct this problem by adding a post-pass that incrementally duplicates any null-yield subtrees updated by the parser. Since a unique maximal sharing of these subtrees does not necessarily exist, this is the only approach that is consistent, correct, and practical. Node reuse strategies can be used to prevent unnecessary recreation of these and other subtrees [25].

4 Resolving Ambiguity

The ultimate use of the abstract parse dag is to enable disambiguation once the needed information is available. This 'filtering' of alternatives can be static (decided at language specification time) or dynamic (decided at program analysis time). Dynamic filtering can involve both syntactic and semantic information. The abstract parse dag and incremental GLR parser together provide a uniform and flexible framework for implementing ambiguity resolution at any point in the analysis process.

4.1 Syntactic Disambiguation

Static syntactic filters, in conjunction with ambiguous grammars, are used frequently in compiler construction. Examples include the operator precedence and associativity specifications in `yacc` and `bison` [1] as well as techniques associated with a particular parse table construction algorithm, such as "prefer shifting". Such methods can be applied at language specification time by selectively removing conflicts from the parse table, and therefore do not result in non-deterministic parsing or multiple representations. Since state-matching incrementalizes transitions in the pushdown automaton, any disambiguation statically encoded in the parse table is supported by the IGLR parser.

When the selection of a preferred interpretation cannot be determined a priori based on the left context and the implicit ('builtin') lookahead, a *dynamic* filter is required. For example, the syntactic ambiguity in C^{++} expressed as "prefer a declaration to an expression" requires a dynamic filter, since competing reductions cannot be delayed until sufficient lookahead has been accumulated [3]. The abstract parse dag allows ambiguities of this form to be encoded using multiple interpretations; an incremental post-pass can then select the preferred structure by directly applying rules such as the one above.[7] Syntactic disambiguation of this form can also take place on-the-fly, provided it occurs only in a deterministic state to avoid contaminating the dynamic lookahead computation. Unlike Ferro and Dion [4], we do *not* retain interpretations eliminated by syntactic filters.

In general, disambiguation specifications [6, 11] can be compiled into a combination of static and dynamic filters. Encoding as much filtering as possible at language specification time decreases both the size of the representation and the analysis time. (This contrasts with existing batch GLR environments, which perform *all* syntactic filtering dynamically [20, 22], and thus require quadratic space for each expression, in contrast to the negligible increases we report in Section 2.1.)

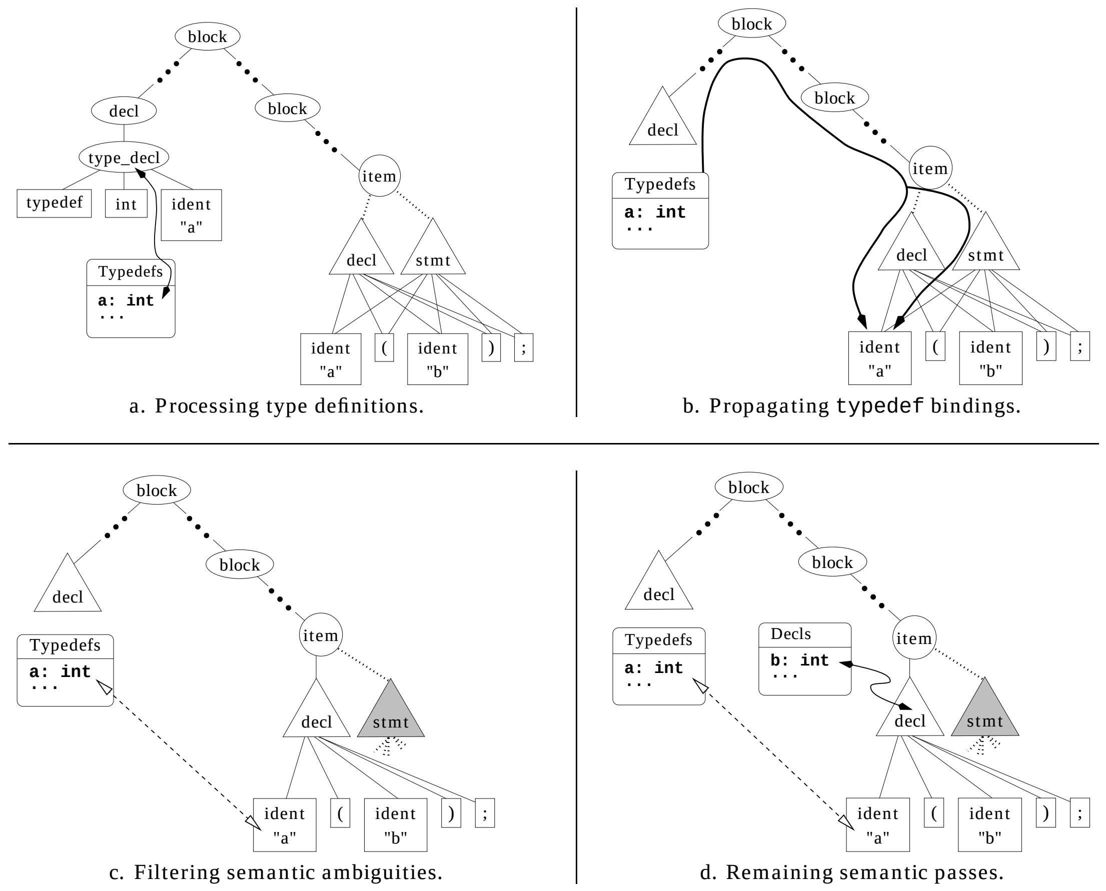
4.2 Semantic Disambiguation

Filters for which the selection criteria are not context-free are referred to as 'semantic' filters. They may be applied in an ad hoc manner or as part of a formal semantic attribution process (using attribute grammars or other approaches). Semantic filters are always dynamic; they are typically applied only after incremental parsing and any syntactic filtering passes have completed. This organization preserves the familiar pass-oriented framework of batch compilation even though the analysis techniques are incremental—it thus avoids the feedback that characterizes the solution to the 'typedef problem' in existing batch systems. While a complete discussion of incremental semantic analysis is beyond the scope of this paper, in this section we briefly outline the sequence of events by which incremental semantic analysis can resolve our running example.

Figure 8 illustrates the sequence of events. After context-free analysis is complete, the first stage of semantic analysis is applied to process `typedef` declarations. Type names introduced by such declarations are gathered into a *binding contour*, which is then propagated throughout the scope. (This information will be inherited by both children of a symbol node, reaching each identifier in an ambiguous region twice.) In a correct program, the binding contour's contents uniquely determine the namespace for each identifier.

With identifier namespaces decided, disambiguation per se can take place: 'parsing' is completed by propagating the namespace decision throughout the ambiguous region. Boolean semantic attributes indicate nodes filtered out of the parse dag in the unwanted interpretation. Since all syntactic and sem-

[7] Contrast this with non-GLR approaches, such as spawning a separate, hand-coded parser for potentially ambiguous regions.



a. Processing type definitions.

b. Propagating `typedef` bindings.

c. Filtering semantic ambiguities.

d. Remaining semantic passes.

Figure 8: Illustration of semantic disambiguation. This shows our running example (using C^{++}, although the situation is similar in both C and Fortran) during the semantic analysis passes. In (a) the basic context-free analysis has been completed, and the first stage of semantic analysis now resolves `typedef` definitions. In (b) this binding information is propagated to the ambiguous regions, allowing the selection of the appropriate namespace for each identifier. In (c) disambiguation per se occurs, as the unwanted interpretation is filtered out (it is retained in case future edits reverse the decision). In (d) semantic analysis continues, using the embedded tree discovered by stages a–c. (Note: The right-hand side of production labels are omitted.)

antic ambiguities have now been resolved, each symbol node can be logically identified with its single remaining child in subsequent passes, allowing tools to treat the result as a normal parse tree.[8]

The order of the passes is the same for both batch and incremental scenarios. In the incremental case, each stage inspects or updates only those portions of the program that have changed or could possibly be affected by preceding changes [16]. An interesting case occurs when a `typedef` declaration is removed: Binding information stored in semantic attributes allows the former uses of the declaration to be efficiently located. At each use site, the interpretation of the ambiguous region will change from a variable declaration to a function call as the namespace of the region's initial identifier is altered. Note that the use sites themselves require no action from the parser; other attributes of the reinterpreted regions are reevaluated as semantic analysis progresses.

[8]Unlike syntactic disambiguation, semantic disambiguation requires that the unwanted interpretations be retained in the abstract parse dag. Semantic filtering uses non-local information (such as declarations in enclosing scopes) that can change and thus require a different resolution *without a change to the local structure*.

4.3 Program Errors

When the program is correct with respect to the language description (and the language as a whole is unambiguous) a single structural representation will eventually be discovered. In the presence of semantic errors, such as missing, malformed, or inconsistent declarations, it may not be possible to determine a single interpretation of the entire structure. In such cases the abstract parse dag maintains multiple interpretations persistently; future edit/analysis cycles may eventually correct the errors and allow the resolution to succeed. These regions are re-evaluated by the parser only when they are modified and by semantic analysis only when they require re-interpretation.

Maintaining every potential interpretation in the presence of an error provides tools in the environment with all relevant information. While the presence of persistent ambiguities may preclude some services, such as code generation, analyses not dependent on the missing information and services that do not require complete resolution (such as presentation) can continue to operate using the unresolved parse dag.

Errors in the context-free syntax may also occur and are detected in the usual fashion: when no parser can successfully shift the (terminal) lookahead symbol. Syntactic error recovery can be supported in the same fashion that we have adopted for deterministic parsing: a history-sensitive, non-correcting strategy that reports deviations from correct program components by integrating only those user modifications that result in at least one valid parse tree. Any modifications remaining are flagged as unincorporated material [27]. This approach is automated, language-independent, and incremental. The primary change needed to support IGLR parsing is an extension of the isolation boundary test to ensure that each non-deterministic region is treated as an atomic unit: partial update incorporation within such a region is not permitted. (This has no practical effect on the efficacy of the recovery, due to the small size of these regions in actual programs.)

5 Implementation and Empirical Performance

The concepts described in this paper have been implemented as part of the *Ensemble* incremental software development environment being prototyped at UC Berkeley. *Ensemble* supports language definition through the off-line compilation of high-level specifications, dynamically loading the compiled language analysis tools into a running environment. Existing language definitions include Java, Modula-2, Fortran, a subset of Lisp, and C (with limited preprocessor support).

The IGLR parser has been implemented in this system as an alternative to the sentential-form parser used for deterministic grammars [25]. The IGLR implementation, which includes the parse table interface but not error recovery code, occupies less than 2000 lines of C++ code, including all tracing and assertion checking. The actual implementation corresponds closely to the algorithm given in Appendix A. Support for abstract parse dags required very little change to *Ensemble's* low-level representation, which is based on the self-versioning document model [26]. Parse table information is produced using a modified version of `bison` that explicitly records all conflicts in the grammar except for those arising from the expansion of the associative sequence notation.

Despite the slightly less efficient stack representation used for GLR parsing relative to deterministic parsing, the IGLR parser performs an initial ('batch') parse nearly as fast as its deterministic counterpart. C,[9] Java, and Modula-2 programs were parsed with both parsers, and yielded an average of 12% overhead due to parsing per se for the deterministic parser, compared with 15% for the IGLR parser. Most of the remaining time was spent in constructing the nodes. In incremental tests (self-cancelling modifications to individual tokens, parsing after each such change) the difference in running times for the two parsers was undetectable.

[9] For this comparison, the 'typedef' ambiguity was removed artificially.

Compared to sentential-form parsing for deterministic grammars, the space consumption of the abstract parse dag is approximately 5% higher, due to the need to record explicit states in the nodes. The difference becomes negligible when semantic attributes, presentation data structures, and other per-node storage is also considered.

The restriction that each non-deterministically parsed region be reconstructed in its entirety whenever it contains at least one edit site imposes little overhead in practice: since none of these regions spanned more than a few nodes in any of our sample programs, the additional reconstruction time was well under 1%, independent of the program, source file, or location of the ambiguous region within the file.

6 Extensions and Future Work

Techniques for expressing both syntactic and semantic filtering in a uniform language would both simplify the language description process and allow optimized performance by applying resolutions at the earliest possible stage. Visser uses priorities and tree patterns to produce static filters [23], but further work is needed.

An integrated model of semantic attribution and dynamic (semantic) filters remains an open problem. It requires extending scheduling algorithms to dags, balancing the restrictions required for efficient static scheduling with sufficient expressive power to model disambiguation methods that arise in practice. This would improve language specifications and enable verification of the combined description.

Incremental, non-deterministic parsing may also find application in rewrite systems and in the iterative analysis of natural language documents.

7 Conclusion

This paper provides a mechanism for applying the open, pass-oriented framework of batch analysis tools to incremental environments. A new IR, the abstract parse dag, is introduced to model ambiguity in programming language analysis. Circular analysis dependencies as they exist in C, C++, Fortran, and other common languages are eliminated by the ability to apply disambiguation filters at any point in the analysis process. Arbitrary CFGs may be used to describe the form of the parse dag, as well as to produce fast incremental parsers based on our IGLR algorithm. Optimal and correct subtree and context sharing in the abstract parse dag are obtained by removing parsing artifacts from the shared parse forest. Empirical measurements demonstrate the space efficiency of our representation and the time efficiency of our reconstruction methods, both of which exploit an underlying language property: localized non-determinism.

8 Acknowledgments

Special thanks to William Maddox for discussing parsing theory and semantic analysis techniques and to John Boyland for his tireless LaTeX assistance and `makebib` tool.

References

[1] A. V. Aho, S. C. Johnson, and J. D. Ullman. Deterministic parsing of ambiguous grammars. *Commun. ACM*, 18(8):441–452, Aug. 1975.

[2] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, Feb. 1970.

[3] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990. Sect. 6.8, 8.1.1.

[4] M. V. Ferro and B. A. Dion. Efficient incremental parsing for context-free languages. In *Proc. 1994 IEEE Intl. Conf. Comp. Lang.*, pages 241–252. IEEE Computer Society Press, May 1994.

[5] Robert Giegerich. Considerate code selection. In Robert Giegerich and Susan L. Graham, editors, *Code Generation — Concepts, Tools, Techniques.*, Workshops in Computing, pages 51–65, Berlin, May 1991. Springer-Verlag.

[6] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. *The syntax definition formalism SDF — Reference Manual*, Dec. 1992.

[7] Paul Hudak et al. Haskell report. *SIGPLAN Not.*, 27(5):R, May 1992.

[8] Fahimeh Jalili and Jean H. Gallier. Building friendly parsers. In *9th ACM Symp. Principles of Prog. Lang.*, pages 196–206, New York, 1982. ACM Press.

[9] Mark Johnson. The computational complexity of GLR parsing. In Masaru Tomita, editor, *Generalized LR Parsing*, pages 35–42. Kluwer Academic Publishers, 1991.

[10] Neil D. Jones and Michael Madsen. Attribute-influenced LR parsing. In U. D. Jones, editor, *Semantics-Directed Compiler Generation*, number 94 in LNCS, pages 393–407, Berlin, 1980. Springer-Verlag.

[11] Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In *Proc. ASMICS Workshop on Parsing Theory*, Milan, Italy, 1994.

[12] Wilf R. LaLonde. Regular right part grammars and their parsers. *Commun. ACM*, 20(10):731–740, 1977.

[13] Marc Lankhorst. An empirical comparison of generalized LR tables. In R. Heemels, A. Nijholt, and K. Sikkel, editors, *Tomita's Algorithm: Extensions and Applications (TWLT1)*, number 91–68 of Memoranda Informatica in Twente Workshops on Language Technology, pages 87–93. Universeit Twente, 1991.

[14] J. M. Larchevêque. Optimal incremental parsing. *ACM Trans. Program. Lang. Syst.*, 17(1):1–15, 1995.

[15] Maryellen C. MacDonald, Marcel Adam Just, and Patricia A. Carpenter. Working memory constraints on the processing of syntactic ambiguity. *Cog. Psych.*, 24(1):56–98, 1992.

[16] William Maddox. *Incremental Static Semantic Analysis*. Ph.D. dissertation, University of California, Berkeley, 1997.

[17] Akira Miyake, Marcel Adam Just, and Patricia A. Carpenter. Working memory constraints on the resolution of lexical ambiguity: Maintaining multiple interpretations in neutral contexts. *J. Memory and Lang.*, 33(2):175–202, Apr. 1994.

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

[19] Luigi Petrone. Reusing batch parsers as incremental parsers. In *Proc. 15th Conf. Foundations Softw. Tech. and Theor. Comput. Sci.*, number 1026 in LNCS, pages 111–123, Berlin, Dec. 1995. Springer-Verlag.

[20] Jan Rekers. *Parser Generation for Interactive Environments*. Ph.D. dissertation, University of Amsterdam, 1992.

[21] Masataka Sassa, Harushi Ishizuka, and Ikuo Nakata. Rie, a compiler generator based on a one-pass-type attribute grammar. *Software—Practice & Experience*, 25(3):229–250, Mar. 1995.

[22] Masaru Tomita. *Efficient Parsing for Natural Languages*. Kluwer Academic Publishers, 1985.

[23] Eelco Visser. A case study in optimizing parsing schemata by disambiguation filters. Technical Report P9507, Programming Research Group, University of Amsterdam, Jul. 1995.

[24] Eelco Visser. Scannerless generalized-LR parsing, 1997. In preparation.

[25] Tim A. Wagner and Susan L. Graham. Efficient and flexible incremental parsing, 1996. Submitted to *ACM Trans. Program. Lang. Syst.*

[26] Tim A. Wagner and Susan L. Graham. Efficient self-versioning documents. In *CompCon '97*, pages 62–67. IEEE Computer Society Press, Feb. 1997.

[27] Tim A. Wagner and Susan L. Graham. Isolating errors—a history-based approach, 1997. In preparation.

[28] David A. Watt. Rule splitting and attribute-directed parsing. In U. D. Jones, editor, *Semantics-Directed Compiler Generation*, number 94 in LNCS, pages 363–392, Berlin, 1980. Springer-Verlag.

Appendix A: IGLR Parsing Algorithm

The non-deterministic component of the IGLR parser is based on Rekers' batch parser [20].

```
class NODE                                        Normal parse dag node
  int type;          production or symbol #
  int state;         deterministic parse state or noState
  setof NODE kids;   rhs of a production; interpretations of a symbol
  NODE (int type, int state, setof NODE kids) {...}

subclass SYMBOL of NODE                           Symbol (choice) node
  SYMBOL (NODE node) {
    type = symbol(node→type);  rule's left-hand side
    state = noState;           multistate by definition
    kids = {node};             first interpretation
  }
  add_choice (NODE node) {kids = kids ∪ node;}

class GSS_NODE                                    Node in the GSS
  int state;          state of constructing parser
  setof LINK links;   links to earlier nodes
  GSS_NODE (int state, LINK link) {...}
  add_link (LINK link) {links = links ∪ link;}

class LINK                                        Edge in the GSS
  GSS_NODE head;  preceding node in the GSS
  NODE node;      parse dag node labeling this edge
  LINK (GSS_NODE head, NODE node) {...}
```

```
bool multipleStates;                              Global variables
NODE shiftLa;  lookahead symbol (subtree)
NODE redLa;    lookahead for reducing
GSS_NODE acceptingParser;
setof GSS_NODE activeParsers, forActor, forShifter;
setof NODE nodes;              production node merge table
setof SYMBOL symbolnodes;  symbol node merge table
```

```
inc_parse (NODE root) {                                  Main routine
  process_modifications_to_parse_dag(root);
  redLa = shiftLa = pop_lookahead(root→bos);
  GSS_NODE gss = new GSS_NODE(0, ∅);
  activeParsers = {gss};
  acceptingParser = ∅;
  multipleStates = false;
  while (acceptingParser == ∅) parse_next_symbol();
  if (shiftLa ≠ eos) recover();
  root→kids[1] = first(acceptParser→links)→node;
  unshare_epsilon_structure(root);
  delete gss;
}
```

```
parse_next_symbol () {                          reduce* shift sequence
  forActor = activeParsers;
  forShifter = nodes = symbolnodes = ∅;
  while (forActor ≠ ∅) do {
    remove a parser p from forActor;
    actor(p); Process all reductions,
  }
  shifter();   then shift.
  redLa = shiftLa = pop_lookahead(shiftLa);
}
```

```
actor (GSS_NODE p) {                            Transition one parser
  while (redLa is an invalid table index)
    redLa = left_breakdown(redLa);
  if (|parse_table[p→state, redLa]| > 1)
    multipleStates = true;
  ∀action ∈ parse_table[p→state, redLa] do
    switch (action) {
      case ACCEPT: if (redLa == eos) acceptingParser = p;
                   else recover();
                   break;
      case REDUCE r: do_reductions(p, r); break;
      case SHIFT s: forShifter = forShifter ∪ <p,s>;
                    break;
      case ERROR: if (activeParsers == ∅)
                  recover(); Recover from a parse error.
    }
}
```

```
shifter () {                                        Shift all parsers
  if (is_terminal(shiftLa) &&
      shiftLa→has_changes(lastParsedVersion))
    relex(shiftLa); Invoke lexer and reset lookaheads
  activeParsers = ∅;
  multipleStates = |for_shifter| > 1;
  while (!is_term(shiftLa) && (multipleStates ||
         forShifter→state ≠ shiftLa→state))
    shiftLa = left_breakdown(shiftLa);
  ∀<q,s> ∈ forShifter do
    if (∃p ∈ activeParsers with p→state == q)
      p→add_link(new LINK(q, shiftLa));
    else activeParsers = activeParser ∪
                    new GSS_NODE(q, new LINK(s, shiftLa))
}
```

```
do_reductions (GSS_NODE p, int rule) {              Find all paths
  GSS_NODE q;
  ∀q such that a path of length arity(rule)
                        from p to q exists do {
    kids = the tree nodes of the links forming the path
                                            from q to p;
    reducer(q, GOTO(q→state, symbol(rule)), rule, kids);
  }
}
```

```
                              Path-restricted version of above function
do_limited_reductions (GSS_NODE p, int rule, LINK link) {
  ∀q such that a path of length arity(rule)
            from p to q through link exists do {
    kids = the tree nodes of the links forming the path
                                            from q to p;
    reducer(q, GOTO(q→state, symbol(rule)), rule, kids);
  }
}
```

```
reducer (GSS_NODE q, int state,                           Perform a
         int rule, setof NODE kids) {              single reduction
  NODE node = get_node(rule, kids, q→state);
  if (∃p ∈ activeParsers with p→state == state)
    if there already exists a direct link from p to q
      add_choice(link→head, node);
    else {
      NODE n = get_symbolnode(node);
      p→add_link(new LINK(q, n));
      ∀m in activeParsers\forActor do
        ∀(reduce rule) ∈ parse_table[m→state, redLa] do
          do_limited_reductions(m, rule, link);
    }
  else {
    GSS_NODE p = new GSS_NODE(state,
                          new LINK(q, get_symbolnode(node)));
    activeParsers = activeParsers ∪ p;
    forActor = forActor ∪ p;
  }
}
```

```
<int,int> cover (setof NODE kids) {                Get offset range
  if (kids == ∅)
    return <offset(shiftLa),offset(shiftLa)>;
  else return <offset(first(kids)),offset(last(kids))>;
}
```

```
NODE get_node (int rule, setof NODE kids,          Create or reuse
              int precedingState) {           a 'production' node
  if (∃n ∈ nodes with n→type == rule && n→kids == kids)
    return n;
  if (multipleStates)
    NODE n = new NODE(rule, noState, kids);
  else NODE n = new NODE(rule, precedingState, kids);
  nodes = nodes ∪ n;
  return n;
}
```

```
add_choice (NODE symnode?, NODE node) {                 Instantiate
  if (symnode? is a symbol node)             symbol nodes lazily
    symnode→add_choice(node);
  else if (symnode? != node) {
    replace symnode? with sym ∈ symbolnodes such that
      first(sym→kids) == symnode?;
    sym→add_choice(node);
  }
}
```

```
NODE get_symbolnode (NODE node) {                Use normal nodes
  if (∃sym ∈ symbolnodes with                whenever possible
       sym→symbol == symbol(node→type) &&
      cover(first(s→kids)→kids) == cover(node→kids))
    sym→add_choice(node);
  else SYMBOL sym = new SYMBOL(node);
  symbolnodes = symbolnodes ∪ sym;
  if (|sym→kids| == 1) return node;   proxy case
  else return sym;                     real case
}
```

The following two routines update the right (input) stack of the incremental parser. One level of structure is removed per invocation of `left_breakdown`. `pop_lookahead` advances the lookahead to the next subtree for consideration by traversing the previous structure of the tree. The previously-parsed version of the program is denoted by `lastParsedVersion`.

```
NODE left_breakdown (NODE n) {
  if (n→arity > 0) {
    n = n→first_child(previousVersion);
    if (n→has_changes(lastParsedVersion))
      return left_breakdown(n);
  } else return pop_lookahead(n);
}
```

```
NODE pop_lookahead (NODE n) {
  while (n→right_sibling(previousVersion) == ∅)
    n = n→parent(previousVersion);
  n = n→right_sibling(previousVersion);
  if (n→has_changes(lastParsedVersion))
    return left_breakdown(n);
  return n;
}
```

The function `process_modifications_to_parse_dag` (called by `inc_parse`) is used to invalidate reductions containing a modified terminal in their yield or implicit (built-in) lookahead. (Structural modifications can also be accommodated.)

Let T denote the set of modified terminals (textual edit sites). Add to T any terminal having lexical lookahead in some $t \in T$. Mark as changed any nonterminal N for which yield(N) $\cup$ the terminal following yield(N) contains any $t \in T$.

Appendix B: Sample C++ Trace

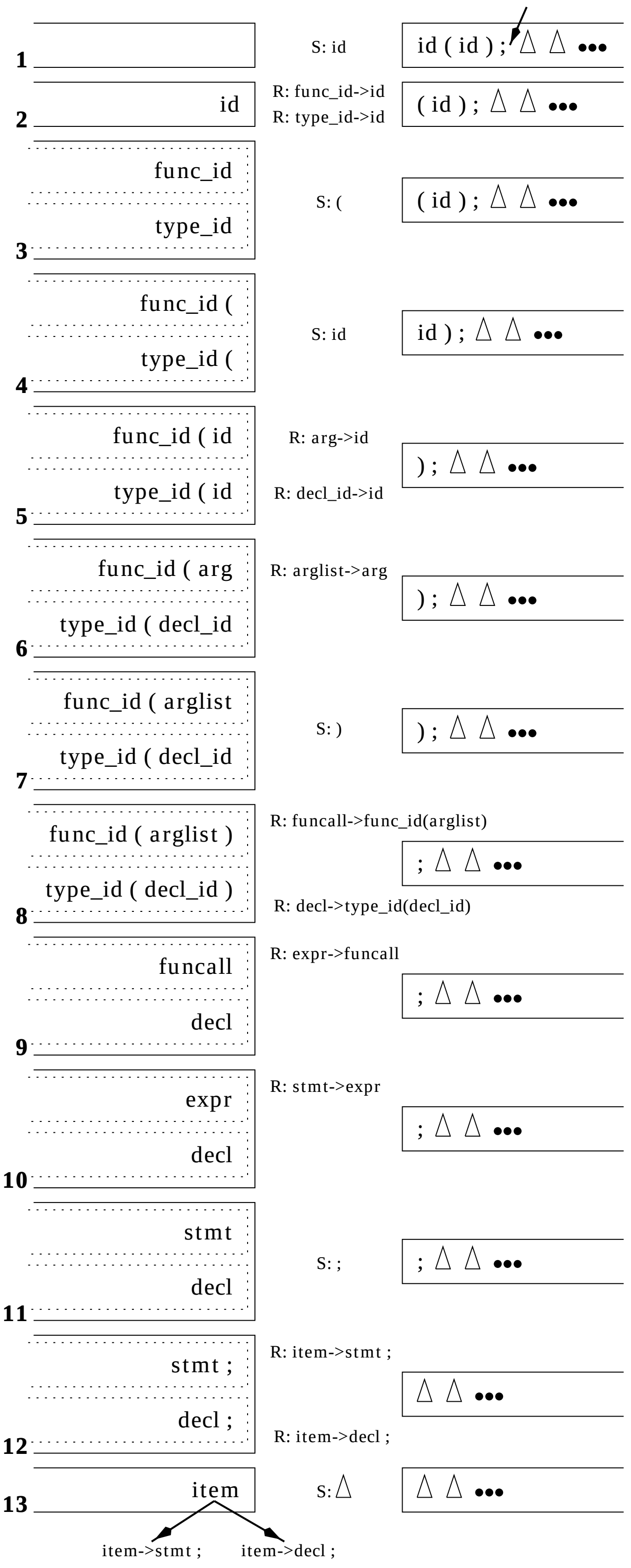
In this example we trace the parser's actions in constructing the dual interpretations of the 'typedef' problem in C++, using a simplified grammar. Consider the input stream as it appears in (1), and suppose the semicolon has been deleted and then reinserted. The region to the left of the semicolon was an ambiguous `item`; the edit to the semicolon causes the parser to discard the non-deterministic structure and read `id(id)` as terminal symbols.

Distinguishing between a normal identifier and a type-name identifier is not context-free; the ambiguity manifests as a reduce/reduce conflict in (2), causing the parser to split. Each of the two parsers now active will create one of the two possible interpretations. A subsequent incremental semantic analysis pass will perform the scope resolution and name binding needed to distinguish the desired interpretation, based on earlier declarations. In a correct program, either a `typedef` or a function declaration will have established the correct namespace for the leading `id`. (The situation would be similar in C, assuming that further input did not yield a purely syntactic resolution.)

While multiple parsers are active, only terminal symbols can be read by the parser. (In this example the breakdown of the ambiguous subtree has already accomplished this.) The breadth-first nature of GLR parsing means that each terminal symbol is shifted in tandem by all active parsers (3, 4, 7, 11).

In (13) context sharing occurs as the two parsers merge into a single parser. The `item` node shown on top of the stack is a symbol node;[10] its two children represent the two interpretations of its terminal yield. Now that the state is once again deterministic, the parser returns to shifting entire subtrees.

[10] Not shown is its lazy instantiation. The first item production serves as a proxy for its symbol node; the attempt to add the second item production as an alterate interpretation forces the installation of a real symbol node. The real symbol node replaces the proxy, which becomes its first child. The second item production becomes the second child.


1
S: id
id (id) ; △ △ ●●●
2
id
R: func_id->id
R: type_id->id
(id) ; △ △ ●●●
3
func_id
type_id
S: (
(id) ; △ △ ●●●
4
func_id (
type_id (
S: id
id) ; △ △ ●●●
5
func_id (id
type_id (id
R: arg->id
R: decl_id->id
) ; △ △ ●●●
6
func_id (arg
type_id (decl_id
R: arglist->arg
) ; △ △ ●●●
7
func_id (arglist
type_id (decl_id
S:)
) ; △ △ ●●●
8
func_id (arglist)
type_id (decl_id)
R: funcall->func_id(arglist)
R: decl->type_id(decl_id)
; △ △ ●●●
9
funcall
decl
R: expr->funcall
; △ △ ●●●
10
expr
decl
R: stmt->expr
; △ △ ●●●
11
stmt
decl
S: ;
; △ △ ●●●
12
stmt ;
decl ;
R: item->stmt ;
R: item->decl ;
△ △ ●●●
13
item
S: △
△ △ ●●●
item->stmt ;
item->decl ;