Efficient and Flexible Incremental Parsing

TIM A. WAGNER and SUSAN L. GRAHAM
University of California, Berkeley

Previously published algorithms for LR(k) incremental parsing are inefficient, unnecessarily restrictive, and in some cases incorrect. We present a simple algorithm based on parsing LR(k) sentential forms that can incrementally parse an arbitrary number of textual and/or structural modifications in optimal time and with no storage overhead. The central role of *balanced sequences* in achieving truly incremental behavior from analysis algorithms is described, along with automated methods to support balancing during parse table generation and parsing. Our approach extends the theory of sentential-form parsing to allow for *ambiguity* in the grammar, exploiting it for notational convenience, to denote sequences, and to construct compact ('abstract') syntax trees directly.

Combined, these techniques make the use of automatically generated incremental parsers in interactive software development environments both practical and effective. In addition, we address *information preservation* in these environments: optimal node reuse is defined, previous definitions are shown to be insufficient, and a method for detecting node reuse is provided that is both simpler and faster than existing techniques. A program representation based on *self-versioning documents* is used to detect changes in the program, generate efficient change reports for subsequent analyses, and allow the parsing transformation itself to be treated as a reversible modification in the edit log.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**]: Programming Environments—*interactive*; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*version control*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *parsing*; *translator writing systems and compiler generators*; E.1 [**Data**]: Data Structures—*trees*

General Terms: Algorithms, Languages, Performance, Theory

Additional Key Words and Phrases: Incremental parsing, optimal reuse, operator precedence, ambiguity, balanced structure, abstract syntax

1. INTRODUCTION

Batch parsers derive the structure of formal language documents, such as programs, by analyzing a sequence of terminal symbols provided by a lexer. Incremental parsers retain the document's structure, in the form of its parse tree, and use this data structure to update the parse after changes have been made by the user or by other tools [Ghezzi and Mandrioli 1980; Wegman 1980; Jalili and Gallier 1982; Ballance et al. 1988; Larchevêque 1995]. Although the topic

This research has been sponsored in part by the Advanced Research Projects Agency (ARPA) under Grant MDA972-92-J-1028, and in part by NSF institutional infrastructure grant CDA-8722788. The content of this paper does not necessarily reflect the position or policy of the U. S. Government.
Authors' addresses: Tim A. Wagner, Reasoning, Inc., 700 E. El Camino Real #300, Mountain View, CA 94040. Susan L. Graham, 771 Soda Hall, Department of Electrical Engineering and Computer Science, Computer Science Division, University of California, Berkeley, CA 94720.
email: twagner@reasoning.com, graham@cs.berkeley.edu
URL: http://http.cs.berkeley.edu/~twagner
http://http.cs.berkeley.edu/~graham.

of incremental parsing has been treated previously, no published algorithms are completely adequate, and most are inefficient in time, space, or both. Several are incorrect or overly restrictive in the class of grammars to which they apply. The central requirement for actual incremental behavior—balancing of lengthy sequences—has been ignored in all previous approaches.[1] Our incremental parser is thus the first to improve on batch performance while reusing existing grammars.

Our incremental parsing algorithm runs in $O(t + s \lg N)$ time for t new terminal symbols and s modification sites in a tree containing N nodes. Performance is determined primarily by the number and scope of the modifications since the previous application of the parsing algorithm. Unlike many published algorithms for incremental parsing, the location of the changes does not affect the running time, and the algorithm supports multiple edit sites, which may include *any combination* of textual and structural updates. The technique applies to any LR-based approach; our implementation uses `bison` [Corbett 1992] and existing grammars to produce table-driven incremental parsers for any language whose syntax is LALR(1).

The parsing algorithm has *no* additional space cost over that intrinsic to storing the parse tree. The algorithm's only requirements are that the parent, children, and associated grammar production of each node be accessible in constant time. *No state information, parse stack links, or terminal symbol links are recorded in tree nodes.* A transient stack is required during the application of the parsing algorithm, but it is not part of the persistent data structure.[2] Our presentation assumes that a complete versioning system exists, since this is necessary in any production environment.

Many parser generators accept ambiguous grammars in combination with additional specifications (e.g., operator precedence and default conflict resolution rules).[3] These techniques provide notational convenience and often result in significantly smaller parse trees, especially in languages like C that are terse and expression dense. We provide new results that allow incremental sentential-form parsing to accommodate ambiguity of this form, preserving both the notational benefits to the grammar and the space-saving properties of the resulting compact trees.

Incremental software development environments (ISDEs) use incremental parsing not just for interactive speed, but because the retained data structure is important in its own right as a shared representation used by analysis, presentation, and editing tools. In this setting, the demands placed on the incremental parsing algorithm involve more than just improved performance

---

[1] Gafter [Gafter 1990] is the notable exception, but his approach precludes possibly-empty sequences, which occur frequently in programming language grammars.

[2] In most systems, nodes will already carry run-time type information. Thus, no additional space is typically required to encode the production represented by a node. In the absence of the history services we describe, two bits per node are needed to track changes made between applications of the parser, and the old value of each structural link must remain accessible until the completion of parsing.

[3] This is essentially a form of parse forest filtering [Klint and Visser 1994] that can be statically encoded so that the parser remains deterministic.

relative to batch systems. It should also provide intelligent *node reuse*: when a structural component (such as a statement) is conceptually retained across editing operations, the parser should not discard and recreate the node representing that component. With intelligent reuse, changes match the user's intuition, the size of the development record is decreased, and the performance of further analyses (such as semantics) improves.

Our incremental parsing algorithm is capable of retaining entire subtrees before, after, and between change points; nodes on a path from the root of the parse tree to a modification site are also reused when doing so is correct and intuitive for the user. Retaining these nodes is especially important since they represent the structural elements (functions, modules, classes) most likely to contain significant numbers of irreproducible user annotations and automated annotations that are time-consuming to restore (such as profile data).

No previously published work correctly describes optimal reuse in the context of arbitrary structural and textual modifications. We present a new formulation of this concept that is independent of the operation of the parsing algorithm and is not limited by the complexity, location, or number of changes. In common cases, such as changing an identifier spelling, our parser makes *no* modifications to the parse tree. Our reuse technique is also simpler and faster than previous approaches, requiring no additional asymptotic time and negligible real time to compute.

The rest of this paper is organized as follows. Section 2 compares previous work on incremental parsing to our requirements and results; it is not needed to understand the material that follows. The program representation and editing model are summarized in Section 3. Section 4 introduces sentential-form parsing and presents an incremental parsing algorithm that uses existing table construction routines. These results are extended in the next section, which develops an optimal implementation of incremental parsing. Support for ambiguous grammars in combination with conflict resolution schemes is covered in Section 6. Section 7 addresses the representation and handling of repetitive constructs (sequences) and constructs a model of incremental performance to permit meaningful comparison to batch parsing and other incremental algorithms. Section 8 develops the theory of optimal node reuse and discusses how reuse computation can be performed in tandem with incremental parsing using the history mechanisms of Section 3. A history-sensitive approach to error recovery has been described separately [Wagner and Graham 1997b].

2. RELATED WORK

Several early approaches to incremental parsing use data structures other than a persistent parse tree to achieve incrementality [Agrawal and Detro 1983; Yeh and Kastens 1988]. While these algorithms decrease the time required to parse a program after a change has been made to its text, they do not materialize the persistent syntax tree required in most applications of incremental parsing.

Some incremental parsing algorithms restrict the user to single-site editing [Reps and Teitelbaum 1989] or to editing of only a select set of syntactic categories [Degano et al. 1988], or can only parse up to the current (single) cursor point [Shilling 1992]. Our goal was to provide an unrestricted editing model that permits mixed textual and structural editing at any number of points (including erroneous edits of indefinite extent and scope) and to analyze the entire program, not merely a prefix or syntactic fragment.

A description of incremental LR(0) parsing suitable for multiple (textual) edit sites was presented by Ghezzi and Mandrioli [1980]. Their algorithm has several desirable characteristics, but its restriction to LR(0) grammars limits its applicability. LL(1) grammars are more practical (having been used in the definitions of several programming languages) and techniques have been developed for incremental top-down parsing using this grammar class [Murching et al. 1990; Beetem and Beetem 1991; Shilling 1992]. Li [1995a] describes a sentential-form LL(1) parser that can accommodate multiple edit sites.

Jalili and Gallier [1982] were the first to provide an incremental parsing algorithm suitable for LR(1) grammars and multiple edit sites and based on a persistent parse tree representation. The algorithm associates parse states with tree nodes, computing the reusability of previous subtrees using *state matching*.[4] This test is sufficient but not necessary, decreasing performance and requiring additional work to compute optimal reuse. (The effect is especially severe for LR(1) grammars, due to their large number of distinct states with equivalent cores.)

More recently, Larchevêque [1995] has extended to LR(k) grammars the *matching condition* originally formulated by Ghezzi and Mandrioli, which allows the parser to retain structural units that fully contain the modification site. His work focuses on the indirect performance gains that accrue from *node reuse* in an ISDE. But unlike the original LR(0) algorithm, this algorithm exhibits linear (batch) performance in many cases. (For example, replacing the opening bracket of a function definition requires reparsing the entire function body from scratch). The definition of node reuse provided does not describe all opportunities for reuse and cannot be considered truly optimal. (It is also linked to the operational semantics of the particular parsing algorithm.) The history mechanisms we define subsume the mark/dispose operations described by Larchevêque.

Petrone [1995] recognizes that explicit states need not be stored in nodes of the parse tree. However, his parsing theory is unnecessarily restrictive; it requires the grammar to be in LR(k) $\cap$ RL(h) for incremental behavior. Grammars outside this class require batch parsing to the right of the first edit in each region (as defined by a matching condition similar to Larchevêque). Node reuse is a subset of that discovered by Larchevêque's algorithm.

Yang [1994] recognizes the utility of sentential-form parsing, but still records parse states in nodes and thus requires a post-pass to relabel subtrees. Li [1995b] describes a sentential-form parser, but his algorithm can generate incorrect parse errors on grammars with ϵ-rules. (It is also limited to complete LR(1) parse tables, since invalid reductions can induce cycling in his algorithm.) Both of these authors suggest 'improving' the parsing algorithm through matching condition checks that actually impede performance and re-

[4]Section 4 reviews state matching.

quire additional space to store the state information in each node.

None of these approaches is ideal. Those that work for unrestricted LR(1) grammars all require additional space in every node of the parse tree (for example, Larchevêque [1995] requires five extra fields per node). Only Degano et al. [1988] address the problem of mixed textual and structural editing, but they then impose a restricted editing framework and require novel table construction techniques. The algorithms that employ matching conditions fail to reuse nodes that overlap modification sites. Existing reuse definitions are suboptimal and tied to the details of particular parsing algorithms. No sentential-form algorithms support ambiguous grammars.

Our approach addresses all these concerns. Our incremental parsing algorithm is based on a simple idea: that a sentential-form LR(1) parser, augmented with reuse computation, can integrate arbitrary textual and structural changes in an efficient and correct manner. Our results are easily extended to enforce any of the restrictions of previous systems, including top-down expansion of correct programs using placeholders, restricting structural editing to correct transformations, and limiting text editing to a subset of nonterminals that must retain their syntactic roles across changes. The technique is suitable for LR(1), LALR(1), SLR(1), and similar grammar classes, and works correctly in the presence of ϵ-rules. The theory extends naturally to LR(k) grammars, although we do not address the general case in the proofs presented here. Existing table construction methods (such as the popular Unix tools `yacc` and `bison`) may be used with very little change. The technique uses less time and space and offers more intrinsic subtree reuse than previous approaches. (Its nonterminal shift check is both necessary and sufficient.) Finally, our approach is designed to provide a *complete* incremental parsing solution: it incorporates a balanced representation of sequences, supports ambiguous grammars and static parse forest filters, and provides provably optimal node reuse.

3. EDITING MODEL AND CHANGE REPORTING

This section reviews our representation of structured documents and a model for editing and transforming them. The object-based versioning services described here provide the incremental parser (and other tools in the environment) with the means both to locate and record modifications. The same interface used to undo textual and structural edits can be used to undo the effects of *any* transformation, including incremental parsing. The representation is based on the self-versioning document model of Wagner and Graham [1997a].

3.1 Representation

The algorithms described in this paper have been embedded in a C++ implementation of the *Ensemble* system developed at Berkeley. *Ensemble* is both a software development environment and a structured document processing system. Its role as a structured document system requires support for dynamic presentations, multimedia components in documents, and high-quality rendering. The need to support software necessitates a sophisticated treatment of structure: fast traversal methods, automated generation mixed with explicit (direct) editing of both structure *and* text, and support for complex incremental transformations [Maverick 1997; Wagner 1997].

Although the *Ensemble* document model supports attributed graphs, in this discussion we will restrict our attention to tree-structured documents, focusing primarily on the text and structure associated with programs. Each document tree is associated with an instance of a *language object*. In the case of programs, this object represents the programming language and contains the grammar and an appropriate specialization of the analysis/transformation tools. The tree's structure corresponds to the concrete or abstract syntax of the programming language; the leaves represent tokens. Tree nodes are instances of strongly-typed C++ classes representing productions in the grammar. These classes are automatically generated when the language description (including the grammar) is processed off-line. Semantic analysis and other tools extend the base class for each production to add their own attributes as slots [Hedin 1992; Maddox 1997].

3.2 Editing Model

We permit an unrestricted editing model: the user can edit any component, in any representation, at any time. These changes typically introduce inconsistencies among the program's components. The frequency and timing of consistency restoration is a policy decision: in *Ensemble*, incremental lexing, parsing, and semantic analysis are performed when requested by the user, which is usually quite frequently but not after every keystroke.[5] Between incremental parses, the user can perform an unlimited number of mixed textual and structural edits, in any order, at any point in the program.[6] The performance of the tools, including the incremental parser, is not adversely affected by the location of the edit sites—changes to the beginning, middle, or end of the program are integrated equally quickly.

Our approach handles all transformations, both user changes and those applied by tools such as incremental parsing, in a uniform fashion. Among other benefits, this allows the user to use existing undo/redo commands to return to any state of the program. This uniform treatment is critical to providing a rational user interface, and requires no additional effort in the implementation of the incremental parser—its effects are captured in the same development history that records all program modifications.

3.3 Program Versions and Change Reporting

An ISDE includes a variety of tools for analyzing and transforming programs. Some tools must be applied in a strict order—for example, semantic analysis cannot be applied until incremental lexing and parsing have restored consistency between the text and structure of a program component. Simple editing

[5] This policy reflects experience showing that re-analysis after every keystroke is unnecessary for adequate performance and the (typically invalid) results would be distracting if presented to the user [Van De Vanter et al. 1992].

[6] There are no restrictions on structural updates save that a node's type remain fixed and that the resulting structure remain a tree. Structural changes not compatible with the grammar *are* permitted; special error nodes are introduced as necessary to accommodate such changes. Textual modifications are represented as local changes to the terminal symbol containing the edit point.

```
bool has_changes([local|nested]).
bool has_changes(version_id, [local|nested]).
```

These routines permit clients to discover changes to a single node or to traverse an entire (sub)tree, visiting only the changed areas. When no version is provided, the query refers to the current version. The optional argument restricts the query to only local or only nested changes.

```
node child(i).
node child(i, version_id).
```

These methods return the i^{th} child. With a single argument, the current (cached) version is used. Similar pairs of methods exist for each versioned attribute of the node: parent, annotations, versioned semantic data, etc.

```
void set_child(node, i).
```

Sets the i^{th} child to `node`. Because the children are versioned, this method automatically records the change with the history log. Similar methods exist to update each versioned attribute.

```
bool exists([version_id]).
```

Determines whether the node exists in the current or a specified version.

```
bool is_new().
```

Determines if a node was created in the current version.

Fig. 1. Summary of node-level interface used by the incremental parser. Each node maintains its own version history, and is capable of reporting both local changes to its attributes and 'nested' changes—modifications within the subtree rooted at the node. The `version_id` arguments refer to the document as a whole; they are efficiently translated into names for values in the local history of each versioned attribute.

operations can also be viewed as transformations, a perspective that is particularly useful when discussing *change reporting*, the means by which tools convey to each other, via the history services, which portion of a program has been changed.

Modifications to the program are initially applied by the user, either directly or through the actions of one or more tools. The completion of a logical sequence of actions is indicated by a *commit* step; once committed, the contents of a version are read-only and are treated as a single, atomic action when changing versions. All versions are named, allowing tools to readily identify any accessible state of the program.

History (versioning) services provide the correspondence between names of versions and values. Their primary responsibility is to maintain the development log, retaining access to 'old' information. Updates to persistent information are routed through the history service, with the current value of versioned data always cached for optimum performance.

The history services also provide a uniform way for tools to locate modifications efficiently. This service is fully general, in that any tool can examine the regions altered between any two versions. Changes can be examined not just at the level of the entire program, but also in a distributed fashion for every node and subtree. This generality is achieved by having each node maintain its own edit history [Wagner and Graham 1997a].

Change reporting is the protocol by which tools discover the modifications

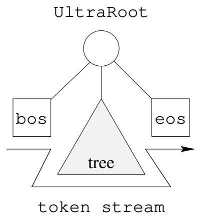


Fig. 2. The relationship between the three permanent sentinel nodes and the parse tree structure. Two permanent tokens bracket the terminal yield of the parse tree, while a third sentinel (the `UltraRoot`) points to both of these tokens as well as the (current) root of the parse tree. The sentinel nodes do *not* change from one version to the next.

of interest to them. Change reporting is mediated by the history service; tools record changes as a side effect of transforming the program and discover changes when they perform an analysis. The history service provides two boolean attributes for each node to distinguish between local and nested changes. *Local changes* are modifications that have been applied directly to a node. For terminal symbols, a local change is usually caused by an operation on the external representation of the symbol. (In the case of programs, local changes usually indicate a textual edit.) Structural editing normally causes local changes to internal nodes. *Nested changes* indicate paths to altered regions of the tree. A node possesses this attribute if and only if it lies on the path between the root and at least one locally-modified node other than itself. Local changes are simply a derived view on the local history log, but nested change annotations must be incrementally computed as synthesized attributes (and must themselves be versioned). Figure 1 summarizes the node-level history interface needed by incremental parsing and node reuse.

Incremental parsing involves three distinct versions of the program:

Reference: A version of the program that represents a parsed state. Any version that concluded with a parse operation may be used; our prototype selects the most recently parsed version as the reference for the subsequent parse. Exception: an initial parse of a newly-entered program has no reference version, since it represents a batch scenario.

Previous: The state of the program immediately prior to the start of parsing. This is the version read by the parser to provide its input stream. (The modifications accrued between the reference version and the previous version determine which subtrees are available for potential reuse.)

Current: The version being written (constructed) by the parser.

Tools in the ISDE, including the incremental parser, use permanent *sentinel* nodes to locate starting points in the mutable tree structure. Three sentinel nodes, shown in Figure 2 are used to mark the beginning and end of the token stream and the root of the tree.

To create a new program, a null tree corresponding to only the sentinels in Figure 2 and a completing production for the start symbol of the grammar is constructed. The initial program text is assigned temporarily as the lexeme of `bos`. Then a (batch) analysis is performed, which constructs the initial version of the persistent program structure; all subsequent structure is derived solely through the incorporation of valid modifications by the parser and other tools.

4. INCREMENTAL PARSING OF SENTENTIAL FORMS

Our incremental parsing algorithm utilizes a persistent parse tree and detailed change information to restrict both the time required to re-parse and the regions of the tree that are affected. The input to the parser consists of both terminal *and nonterminal* symbols; the latter are a natural representation of the unmodified subtrees from the reference version of the parse tree. We begin by discussing tests for subtree reuse, then present a simplified algorithm for incremental parsing that introduces the basic concepts. Section 5 extends these results to achieve optimal incrementality; subsequent sections discuss representation issues and additional functionality.

4.1 Subtree Reuse

Many previous algorithms for incremental parsing of LR(k) or LALR(k) grammars have relied on *state matching*, which incrementalizes the push-down automata of the parser. The configuration of the machine is summarized by the current parse state, and each node in the parse tree records this state when it is shifted onto the stack. To test an unmodified subtree for reuse at a later time, the state recorded at its root is compared to the machine's current state. If they match, and any required lookahead items are valid, then the parser can shift the subtree without inspecting its contents. Testing the validity of the lookahead is usually accomplished through a conservative check: the k terminal symbols following the subtree on the previous parse are required to follow it in the new version as well.

One disadvantage of state matching is the space associated with storing states in tree nodes. State matching also restricts the set of contexts in which a subtree is considered valid, since the state-match test is sufficient but not necessary. The overly restrictive test is particularly limiting with LR(1) parse tables, as opposed to LALR(1), because the large number of similar distinct states (i.e., distinct item sets with identical cores) practically guarantees that legal syntactic edits will not have valid state matches. The failure to match states for a subtree in a grammatically correct context causes a state-matching incremental parser to discard the subtree and rebuild an isomorphic one labeled with different state numbers. LALR(1) parsers fare better with state-matching algorithms because a greater proportion of modifications permit the test to succeed.

Sentential-form parsing is a strictly more powerful technique than state matching for deterministic grammars, capturing more of the 'intrinsic' incrementality of the problem. For LR(0) parsers, the mere fact that the grammar symbol associated with a subtree's root node can be shifted in the current parse state indicates that the entire subtree can be incorporated without further analysis. The situation is similar, though more complex, for the LR(1) case. Stated informally, the fact that a subtree representing a nonterminal is shiftable in the current parse state means that the entire subtree except for its right-hand edge (the portion affected by lookahead outside the subtree) can be immediately reused. Sentential-form parsing provides incrementality without the limitations of state matching: no states are recorded in nodes, subtrees

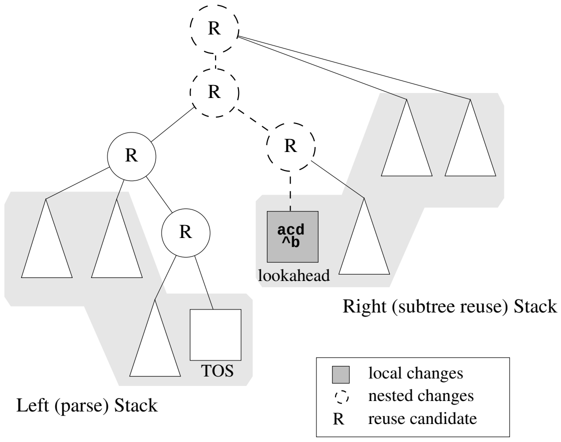


Fig. 3. Incremental parsing example. This figure illustrates a common case: a change in the spelling of an identifier results in a 'split' of the tree from the root to the token containing the modified text. The shaded region to the left becomes the initial contents of the parse stack, which is instantiated as a separate data structure because it contains a mixture of old and new subtrees. The shaded region to the right provides the potentially reusable portion 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. Except when new text is being scanned, the top element of the right stack serves as the parser's lookahead symbol. The remaining nodes in this figure are all candidates for explicit reuse (Section 8). In the example shown, the tree will be 'sewn up' along the path of nested changes; the parser will not need to create any new nodes to incorporate this change to the program.

can be reused in any grammatically-correct context, and lookahead validation is accomplished 'for free' by consuming the input stream.

Like Jalili and Gallier, we conceptually 'split' the tree in a series of locations determined by the modifications since the previous parse. Modification sites can be either interior nodes with structural changes or terminal nodes with textual changes,[7] and the split points are based on the (fixed) number of lookahead items used when constructing the parse table. The input stream to the parser will consist of both new material (in the form of tokens provided by the incremental lexer) and reused subtrees; the latter are conceptually on a stack, but are actually produced by a directed traversal over the previous version of the tree. An explicit stack is used to maintain the new version of the tree while it is being built. This stack holds both symbols (nodes) and states (since they are not recorded within the nodes). Figure 3 illustrates a common case, where a change in identifier spelling has resulted in a split to the terminal symbol containing the modified text.

We now formalize the concept of shifting subtrees.

Notation. Let t_i denote a terminal symbol and X_i an arbitrary symbol in the (often implicit) grammar G. Greek letters denote (possibly empty) strings of symbols in G. k denotes the size of the terminal lookahead used in constructing

[7] All textual and structural modifications are reflected in the tree itself. Section 3 discusses the representation of programs and the techniques for summarizing changes.

the parse table. s_i denotes a state. Subscripts indicate left-to-right ordering. $LA(s_i)$ denotes the union of the lookahead sets for the collection of LR(1) items represented by s_i. $GOTO(s_i, X)$ indicates the transition on symbol X in state s_i. (This is *not* a partial function; illegal transitions are denoted by a distinguished error value.) We use additional terminology from Aho et al. [1986].

THEOREM 4.1.1.
Consider a conventional batch LR(1) *parser in the configuration:*

$$\boxed{\ldots s_0 X_1 s_1 X_2 s_2 \ldots s_{n-1} X_n s_n} \quad \boxed{t_1 t_2 \ldots t_m t_{m+1} \ldots}$$

Suppose $A \stackrel{*}{\Rightarrow} t_1 \ldots t_m$ $(m \geq 0)$. *Note that* A *may derive the empty string* (ϵ).

If $GOTO(s_n, A) = s_i$ *and* $t_{m+1} \in LA(s_n)$, *then the parser will eventually enter the configuration:*

$$\boxed{\ldots s_0 X_1 s_1 X_2 s_2 \ldots s_{n-1} X_n s_n\, A\, s_i} \quad \boxed{t_{m+1} \ldots}$$

PROOF. By the correctness of LR(1) parsing and the fact that $X_1 \ldots X_n A t_{m+1}$ is a viable prefix. □

The results of Theorem 4.1.1 cannot be used directly: testing whether the terminal symbol following a subtree is in the lookahead set for the current state is not supported by existing parse tables, even though such information is available during table construction. Instead, we use this result in a more restricted fashion.[8]

If a subtree has no internal modifications and its root symbol is shiftable in the current parse state, then all parse operations up to and including the shift of the final terminal symbol in the tree are pre-determined, and we can put the parser directly into that configuration, without additional knowledge of legal lookaheads. This transition is actually accomplished by shifting the subtree onto the parse stack, then removing ('breaking down') its right edge (Figure 4). The situation is complicated slightly by the possibility that one or more subtrees with null yield may need to be removed from the top of the parse stack as well, since they also represent reductions predicated on an uncertain lookahead.[9] Figure 5 illustrates the breakdown procedure. Its validity is established by the following theorem, which relates the configuration (parse stack contents and parse state) of a batch parser to that of an incremental parser that has shifted a nonterminal and then invoked `right_breakdown`.

THEOREM 4.1.2.
Let $A \stackrel{*}{\Rightarrow} t_1 \ldots t_m$, $m \geq 1$ *be a production in* G, *and* $X_1 X_2 \ldots X_n A$ *a viable prefix. Let* B *denote a (batch)* LR(1) *parser for the grammar* G *in the configuration:*

$$\boxed{\ldots s_0 X_1 s_1 X_2 s_2 \ldots s_{n-1} X_n s_n} \quad \boxed{t_1 t_2 \ldots t_m \ldots}$$

[8]In Section 5 we describe a technique that involves minimal changes to table construction methods and provides better incremental performance than terminal lookahead information could achieve.
[9]Right-edge breakdowns are done *eagerly* to avoid cycling when the parse table contains default reductions or is not canonical (e.g., is LALR(1) instead of LR(1)) and the input is erroneous. If complete LR(k) tables are used, right-edge breakdowns can be done on demand, in an analogous fashion to the left-edge breakdown shown in Figure 7.

```
Remove any subtrees on top of parse stack with null yield, then
break down right edge of topmost subtree.
right_breakdown () {
  NODE *node;
  do { Replace node with its children.
    node = parse_stack→pop();
    Does nothing when child is a terminal symbol.
    foreach child of node do shift(child);
  } while (is_nonterminal(node));
  shift(node); Leave final terminal symbol on top of stack.
}

Shift a node onto the parse stack and update the current parse state.
void shift (NODE *node) {
  parse_stack→push(parse_state, node);
  parse_state = parse_table→state_after_shift(parse_state, node→symbol);
}
```

Fig. 4. Procedures used to break down the right-hand edge of the subtree on top of the parse stack. On each iteration, node holds the current top-of-stack symbol. Any subtree with null yield appearing in the top-of-stack position is removed in its entirety.

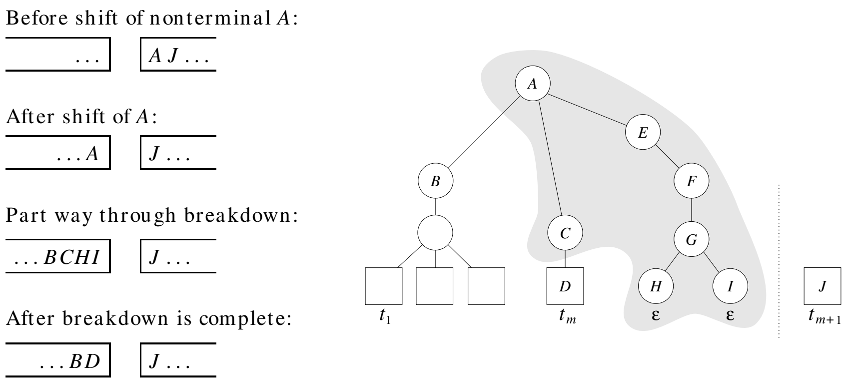


Fig. 5. Illustration of right_breakdown. The shaded region shows the reductions 'undone' by the breakdown—all nodes representing reductions predicated on the following terminal symbol (J) are removed. Any subtrees with null yield are discarded, then the right-hand edge of the subtree on top of the stack is removed, leaving its final terminal symbol in the topmost stack position. (The parse stack holds both states and nodes; only node labels are shown here.)

Let I denote an incremental LR(1) *parser for the grammar G in the configuration:*

$$\boxed{\ldots s_0 X_1 s_1 X_2 s_2 \ldots s_{n-1} X_n s_n} \quad \boxed{A \ldots}$$

where yield$(A) = t_1 \ldots t_n$. *The configuration of I following a shift of A and subsequent invocation of the breakdown procedure (Figure 4) is identical to the configuration of B immediately after it shifts* t_m.

PROOF. Each iteration of the loop in `right_breakdown` leaves a viable prefix on I's stack. At the conclusion of the routine, t_m will be the top element. Since the parse tree for the derivation of A is unique, I's final stack configuration must match that of B; the equivalence of the parse states follows. □

4.2 An Incremental Parsing Algorithm

We now use Theorem 4.1.2 to construct an incremental parser. Pseudocode for this algorithm is shown in Figure 6.

The algorithm in Figure 6 represents a simple 'conservative' style of incremental parsing very similar to a state-matching algorithm. The input stream is a mixture of old subtrees (from the previous version of the parse tree) that is constructed on the fly by traversing the previous tree structure using the local/nested change information described in Section 3. The parse stack contains both states and subtrees, and is discarded when parsing is complete. Incremental lexing can either be performed in a separate pass prior to parsing or, as shown here, in a demand-driven way as the incremental parser encounters tokens that may be inconsistent. We assume that the incremental lexer resets the lookahead (`la`) to point to the next old subtree when it completes re-lexing of a contiguous section.

Reductions occur as in a conventional batch parser, using a terminal lookahead symbol to index the parse table. Shifts, however, may be performed using non-trivial subtrees representing nonterminals. Unlike state matching, the shift test is not only sufficient but also necessary: a valid shift is determined based on the grammar, not the relationship between two configurations of the parse stack.

Subtrees that cannot be shifted are broken down, one level at a time, as if they contained a modification. After a non-trivial subtree is shifted, all reductions predicated on the next terminal symbol are removed by a call to `right_breakdown`. (These reductions are often valid, in which case the discarded structure will be immediately reconstructed. In the following section we eliminate this and other sources of sub-optimal behavior.)

The correctness of this algorithm is based on Theorem 4.1.2, which associates the configuration of the incremental parser immediately prior to each reduction with a corresponding configuration in a batch parser.

Theorems 4.1.1 and 4.1.2 apply equally well to LALR(1) and SLR(1) parsers, so the algorithm given in Figure 6 can be used for these grammar classes and parse tables as well. The only restriction, which applies to *any* grammar class, is that table construction techniques cannot use lossy compression

```
void inc_parse () {
  Initialize the parse stack to contain only bos.
  parse_stack→clear(); parse_state = 0; parse_stack→push(bos);
  NODE *la = pop_lookahead(bos); Set lookahead to root of tree.
  while (true)
    if (is_terminal(la))
      Incremental lexing advances la as a side effect.
      if (la→has_changes(reference_version)) relex(la);
      else
       switch (parse_table→action(parse_state, la→symbol)) {
           case ACCEPT:   if (la == eos) {
                            parse_stack→push(eos);
                            return; Stack is [bos start_symbol eos].
                          } else {recover(); break;}
         case REDUCE r: reduce(r); break;
         case SHIFT s:  shift(s); la = pop_lookahead(la); break;
         case ERROR:    recover(); break;
         }
    else this is a nonterminal lookahead.
      if (la→has_changes(reference_version)
        la = left_breakdown(la); Split tree at changed points.
      else {
       Reductions can only be processed with a terminal lookahead.
       perform_all_reductions_possible(next_terminal());
       if (shiftable(la))
         Place lookahead on parse stack with its right-hand edge removed.
         {shift(la); right_breakdown(); la = pop_lookahead(la);}
       else la = left_breakdown(la);
      }
}
```

Fig. 6. An incremental parsing algorithm based on Theorem 4.1.2. The input is a series of subtrees representing portions of the previous parse tree intermixed with new material (generated by invoking the incremental lexer whenever a modified token is encountered). After each nonterminal shift, `right_breakdown` is invoked to force a reconsideration of reductions predicated on the next terminal symbol. Non-trivial subtrees appearing in the input stream are broken down when the symbol they represent is not a valid shift in the current state or when they contain modified regions. `next_terminal` returns the earliest terminal symbol in the input stream; when the lookahead's yield is not null, this will be the leftmost terminal symbol of its yield. The `pop_lookahead` and `left_breakdown` methods are shown in Figure 7; `has_changes` is a history-based query from Figure 1. `bos` and `eos` are the token sentinels illustrated in Figure 2.

Decompose a nonterminal lookahead.
```
NODE *left_breakdown (NODE *la) {
  if (la→arity > 0) {
    NODE *result = la→child(0, previous_version);
    if (is_fragile(result)) return left_breakdown(result);
    return result;
  } else return pop_lookahead(la);
}
```

Pop right stack by traversing previous tree structure.
```
NODE *pop_lookahead (NODE *la) {
  while (la→right_sibling(previous_version) == NULL)
    la = la→parent(previous_version);
  NODE *result = la→right_sibling(previous_version);
  if (is_fragile(result)) return left_breakdown(result);
  return result;
}
```

Fig. 7. Using historical structure queries to update the right (input) stack in the incremental parser. The lookahead subtree is decomposed one level for each invocation of `left_breakdown`, conceptually popping the lookahead symbol and pushing its children in right-to-left order (analogous to one iteration of `right_breakdown`'s loop). `pop_lookahead` advances the lookahead to the next subtree for consideration, using the previous structure of the tree. The boxed code is used to support ambiguous grammars (Section 6).

on the *GOTO* table (it cannot be rendered as a partial map). While only legal nonterminal shifts arise in batch parsing, as the final stage in a reduction, sentential-form parsing needs an *exact* test to determine whether a given subtree in the input can be legally shifted.

To establish the running time of the algorithm in Figure 6,[10] suppose that the height of a subtree containing N nodes is $O(\lg N)$. If there are s modification sites, the previous version of the tree will be split into $O(s \lg N)$ subtrees. Tokens resulting from newly inserted text are parsed in linear time. When the lookahead symbol is a reused subtree, $O(\lg N)$ time is required to access its leading terminal symbol in order to process reductions. If the subtree can be shifted in the new context, $O(\lg N)$ time is also consumed in reconstructing its trailing reduction sequence using `right_breakdown`. If we assume that each change has a bounded effect (results in a bounded number of additional subtree breakdowns), then the combined cost of shifting a nonterminal symbol is $O(\lg N)$. For t new tokens, this yields a total running time of $O(t + s(\lg N)^2)$.

Note that there is no persistent space cost attributable solely to the incremental parsing algorithm, since the syntax tree is required by the environment. The ability to shift subtrees independent of their previous parsing state avoids the need to record state information in tree nodes.

[10]Section 7 discusses the model of incremental parsing and the assumptions regarding the form of the grammar and parse tree representation.

5. OPTIMAL INCREMENTAL PARSING

The previous section developed an incremental parsing algorithm that used existing information in LR (or similar) parse tables. In this section we improve upon that result by avoiding unnecessary calls to `right_breakdown` and by eliminating the requirement that only terminal symbols can be used to perform reductions. The result is an optimal algorithm for incremental parsing, with a running time of $O(t + s \lg N)$. (We focus primarily on the $k = 1$ case, but also indicate how additional lookahead can be accommodated.)

The algorithm in Figure 6 can perform reductions only when the lookahead symbol is a terminal; when the lookahead is a nonterminal, that algorithm must traverse its structure to locate the leading terminal symbol. By providing slightly more information in the parsing tables however, we can use nonterminal lookaheads to make reduction decisions *directly*, eliminating one source of the extra $\lg N$ factor without maintaining a 'next terminal' pointer in each node.

When the lookahead symbol is a nonterminal with non-null yield that extends the viable prefix, there is no need to access the leftmost terminal symbol of the yield in order to perform reductions: If Z can follow Y in a rightmost derivation and $Z \stackrel{*}{\Rightarrow} t_1 \ldots t_m (m \geq k)$, then a reduction of a handle for Y by a batch parser when the first k symbols of Z constitute the lookahead can be recorded in the parse table as the action to take with the nonterminal lookahead Z.[11] This change avoids the performance cost of extracting the initial k terminals from the subtree representing the current lookahead symbol. Only when the lookahead is invalid (does not extend the viable prefix) or contains modifications must it be broken down further.

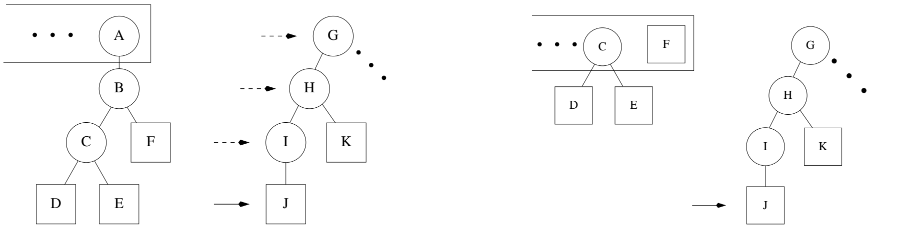
Basing reductions on nonterminal lookaheads is not itself sufficient to improve the asymptotic performance results of the previous section's algorithm; unnecessary invocations of `right_breakdown` must also be eliminated. Spurious reconstructions can be avoided by parsing *optimistically*: the parser omits the call to `right_breakdown` after shifting a subtree, and performs reductions even when the lookahead contains fewer than k terminal symbols in its yield. When such actions turn out to be correct, unnecessary work has been avoided. If one or more actions were incorrect, the problem will be discovered before k terminal symbols past the point of the invalid action have been shifted. The parser backtracks efficiently from invalid transitions; in the $k = 1$ case, backtracking is merely a delayed invocation of `right_breakdown`.[12] Optimistic behavior thus improves both the asymptotic and the practical performance of the incremental parser.

The algorithm in Figure 8 implements the optimistic strategy by a technique similar to the *trial parsing* used in batch parser error recovery [Burke and Fisher 1987]. Suppose we can legally shift a reused subtree, and, *in the*

[11] The change to existing table generators is minor: the parse table must be augmented slightly to represent *all* valid (and invalid) nonterminal transitions explicitly. Algorithms for constructing parse tables for the classes of parsers described here [Aho et al. 1986] are easily modified to enumerate all lookahead symbols rather than terminals alone.
[12] If the grammar contains V nonterminals, the amount of backtracking is limited to $O(kV)$.

```
void inc_parse () {
  bool verifying = false;
  Initialize parse stack to contain only bos.
  parse_stack→clear(); parse_state = 0; parse_stack→push(bos);
  NODE *la = pop_lookahead(bos); Set lookahead to first subtree following bos.
  while (true)
    if (is_terminal(la))
      if (la→has_changes(reference_version)) relex(la);
      else
        switch (parse_table→action(parse_state, la→symbol)) {
          case ACCEPT:   if (la == eos) {
                           push(eos); return; Stack is [bos start_symbol eos].
                         } else {recover(); break;}
          case REDUCE r: verifying = false; reduce(r); break;
          case SHIFT s:  verifying = false; shift(la);
                         la = pop_lookahead(la); break;
          case ERROR:    if (verifying) {
                           right_breakdown(); Delayed breakdown.
                           verifying = false;
                         }
                         else recover(); Actual parse error.
      }
    else this is a nonterminal lookahead.
      if (la→has_changes(reference_version))
        la = left_breakdown(la); Split to changed point.
      else
        switch (parse_table→action(parse_state, la→symbol)) {
          case REDUCE r: if (yield(la) > 1) verifying = false;
                         reduce(r); break;
          case SHIFT s:  verifying = true; shift(la);
                         la = pop_lookahead(la); break;
          case ERROR:    if (la→arity > 0) la = left_breakdown(la);
                         else la = pop_lookahead(la);
      }
}
```

Fig. 8. An improved incremental parsing algorithm. Its correctness is expressed by Theorem 5.1.1. It works on any LR(1) or LALR(1) table in which the set of nonterminal transitions is both complete and correct. The boxed statements are included *only* when canonical LR(k) tables are used; they improve performance slightly by also validating *reductions* when the lookahead symbol is not an ϵ-subtree. Since all parsing classes we consider have the viable prefix property, the ability to shift any non-ϵ-subtree automatically validates any speculative reductions, including speculatively-shifted ϵ-subtrees. If a real parse error occurs, the algorithm invokes `recover()` in the same configuration in which a batch parser would initiate recovery of the error.



9a. Discarding left edge of right stack. 9b. Discarding right edge of left stack.

Fig. 9. The situation when an error is detected. The first course of action is to progressively traverse the left edge of the subtree being considered for reuse (a). This action is the counterpart to `right_breakdown`, except that it is implemented incrementally simply by changing the lookahead item on top of the right stack. If an error persists with a terminal symbol in the lookahead position, then `right_breakdown` is used to ensure that the topmost element of the parse stack is also a terminal symbol. If the input is correct, parsing will continue as usual from this point. In the event an actual parse error exists, one or more invalid reductions will typically be (re)performed at this point unless the parse tables are canonical. In any event, the error will be detected before any further terminal symbols are shifted, and error recovery will be initiated in *exactly* the same configuration as in a batch parser.

resulting state, can continue by shifting additional symbols deriving at least one terminal symbol (or incorporating the end of the input). The only way this can happen is if the first subtree was correct in its entirety, *including* its final reduction sequence. Further shifts of k terminal symbols indicate they were in the lookahead set after shifting the initial subtree, proving that any reductions optimistically retained (or applied based on insufficient lookahead) were indeed valid.[13] These reductions include any subtrees with null yield (ϵ-subtree) on top of the parse stack, as well as the right-hand edge of the topmost non-ϵ-subtree.

With this optimistic strategy, several possibilities obtain when the lookahead symbol does not indicate a shift or reduce action. (Figure 9 illustrates the sequence of events.) The parser begins by incrementally discarding structure in a nonterminal lookahead until either a valid action is indicated by the parse table or the lookahead is a terminal symbol. At that point, if the error persists, the algorithm uses `right_breakdown` to discard speculative (unverified) reductions. At this point the top of the parse stack and the lookahead are both terminal symbols. If the input is valid, the incremental parser proceeds to shift it after zero or more (valid) reductions.

In the event of an actual parse error, the algorithm of Figure 8 invokes error handling in exactly the same configuration where a batch parser would discover the error. For canonical tables, this configuration will have a terminal symbol on the top of the parse stack and in the lookahead position. For other classes of parsers, one or more invalid reductions may be performed before

[13]The $k > 1$ case is complicated by the fact that up to $k - 1$ terminal symbols can be shifted before the error is discovered.

the error is detected.[14] (When this happens, note that setting `verifying` to `false` is essential to prevent the incremental parser from cycling. Otherwise the invalid reductions would be re-applied, only to be followed once again by a call of `right_breakdown`.)

When using canonical LR(1) tables, *reductions* based on a non-ϵ-subtree lookahead can be used to validate speculative reductions. Lossy compression of the terminal reduction actions and the invalid reductions permitted by other parser classes (LALR(1), SLR(1)) limit validation to shifts of non-ϵ-subtrees.[15] However, if an LALR or SLR parser generator identifies reductions guaranteed never to be erroneous, reduction validation can be employed on a case-by-case basis.

The flow of control in Figure 8 is similar to that of the previous algorithm, except for the two optimizations defined above. In simple cases, such as the example illustrated in Figure 3, each subtree appearing in the input stream is shifted with no breakdowns except for those required to expose the modification sites. The conservative invocation of `right_breakdown` after each nonterminal shift has been replaced by the `ERROR` cases, which use `right_breakdown` to implement backtracking. The ability to reduce on a nonterminal lookahead results in a new `REDUCE` case that is identical to its terminal counterpart.

5.1 Correctness

We now demonstrate that the parse tree produced by the algorithm in Figure 8 is the same as the parse tree resulting from a batch parse using an identical parse table, thus establishing correctness in the $k = 1$ case.[16]

First, we justify the optimized shifting strategy. Recall that Theorem 4.1.1 does not apply to the set of reductions removed by `right_breakdown`. But if the parser can continue by shifting (or, in the case of a canonical parse table, reducing) using a non-ϵ-subtree lookahead, then clearly the configuration immediately after the shift represented a valid prefix of a rightmost derivation and the use of `right_breakdown` was unnecessary. Since the lookahead is known to be valid, Theorem 4.1.2 ensures that the configuration of the batch and incremental parsers are identical after the shift operation, even without the breakdown procedure.

Second, consider making parsing actions on the basis of a lookahead symbol represented by an ϵ-subtree in the input stream. Since the length of the terminal yield of the lookahead is less than k, any decisions based on it are potentially invalid. Three cases can arise:

—In the case of an error, `left_breakdown(la)` is invoked. Eventually either a non-ϵ-subtree lookahead is reached or one of the cases below applies.

[14]Replacing error entries in the terminal transition portion of a canonical parse table with reductions has the same effect.
[15]All the parsing classes we consider here retain the viable prefix property when $k = 1$, which ensures that a shift of a non-ϵ-subtree is possible only if the preceding reduction sequence was valid.
[16]The general case is similar, but bookkeeping in the proof, as in the algorithm, is more complex due to the need to backtrack after shifting a non-ϵ-subtree.

—In the case of a `REDUCE` action, a new node is created without advancing the lookahead.
—In the case of a `SHIFT` action, the ϵ-subtree is pushed onto the parse stack. (This is equivalent to the application of one or more reductions.)

Shifting or reducing based on an ϵ-subtree lookahead merely adds to the set of pending reductions. No subsequent shift of a non-ϵ-subtree can occur unless it extends a viable prefix; if *any* of the reductions are invalid, an eventual call to `right_breakdown` will remove the entire reduction sequence and apply the correct set of reductions using the following terminal symbol.

We can now establish that the configuration of this parser is identical to that of a batch parser at a number of well-defined *match points*.

THEOREM 5.1.1.
The configuration of the incremental parser defined by the algorithm in Figure 8 matches that of a batch parser using the same parse table information in the following cases:

(1) At the beginning of the parse, with an empty stack and the lookahead set to `bos`.
(2) At the end of the parse, when the `accept` *routine is invoked.*
(3) When an error is detected (and the `recover` *routine is invoked).*
(4) Immediately prior to a shift of any non-ϵ-subtree by the incremental parser.

PROOF SKETCH. Equality clearly holds in the first case. (2) can be modeled as a special case of (4) by treating it as a 'shift' of one or more end-of-stream (`eos`) symbols in order to reduce to the start symbol of an augmented grammar. The argument for (3) has already been presented. Case (4) relies on the argument for optimized shifting in conjunction with backtracking as presented above, observing that the incremental parser has performed all possible reductions when a shift is about to occur. □

COROLLARY 5.1.2.
The incremental parsing algorithm of Figure 8 produces the same parse tree constructed by a batch parser reading the same terminal yield.

5.2 Optimality

We now investigate the claim that the algorithm in Figure 8 (algorithm A) is optimal with respect to a general model of incremental shift/reduce parsing. We do this by establishing that no other algorithm A' of this form can improve asymptotically on the total number of steps, independent of the grammar and edit sequence. First, assume as input

—A sequence of reused subtrees and new tokens; the reused subtrees are provided by a traversal of the changed regions of the previous version of the tree.
—A parse table for a grammar G, in which nonterminal transitions are both complete and correct.

We use the conventional model of shift/reduce parsing, augmented with the ability to shift nonterminals in the form of non-trivial subtrees retained from the previous version of the tree. A node may be reused in the new version of the tree if its child nodes are identical in both trees. (Section 8 explores models of node reuse in greater detail.) The cost model charges $O(1)$ time for each node visited or constructed.

As stated previously, our version of sentential-form parsing uses a subtree shift test that is both necessary and sufficient. It follows immediately that no other parsing algorithm can perform fewer shifts.

To understand why the number of reductions is asymptotically optimal, we first consider a restricted case: LR(1) parse tables in which the terminal action transitions are also complete (i.e., no use of 'default reductions'). We also make a straightforward replacement of the `right_breakdown` routine in Figure 4 with one that operates in a stepwise fashion. Now assume some other algorithm A' avoids a reduction that our algorithm performs. Since the reduction can be avoided by A', it must reuse a node N to represent the same reduction in both trees. If N was marked with nested changes prior to parsing, then the cost of the extra reduction is asymptotically subsumed by the traversal needed to generate the input to the algorithm. Otherwise N was broken down by `left_breakdown` unnecessarily in order to trigger one or more calls to `right_breakdown`. But the order in which reductions are reconsidered when two non-trivial subtrees adjacent in the input stream cannot be adjacent in the new tree is arbitrary: without additional knowledge, no algorithm can choose an optimal order for these tests a priori. Thus some different combination of grammar and edit sequence must result in A' requiring more reductions than our algorithm.

Now suppose a parser class that permits erroneous reductions and/or lossy compression of terminal reduction actions in the parse table, along with the version of `right_breakdown` shown in Figure 4. In this case, a reduction performed by A' and not by our algorithm may also be due to the fact that `right_breakdown` removes a reduction unnecessarily. (Recall that this routine must assure a configuration in which a terminal symbol is on top of the parse stack in order to avoid cycling in the presence of erroneous reductions.)

First, suppose A' predicates a node's reusability (in part) on the lookahead symbol, as is done in state-matching approaches. Since the two subtrees in question were not necessarily adjacent in the previous version of the parse tree, we can easily exhibit grammars and edit sequences in which A' performs more reductions than A by arranging for the following terminal symbol to be different than in the previous parse tree.

Now suppose A' does *not* predicate reusability on the lookahead symbol. To avoid configurations in which A invokes `right_breakdown`, A' must either shift sub-optimally or remove reductions unnecessarily in some circumstances. If A' *does* enter such a configuration, then to avoid cycling it must remove all reductions dependent upon the lookahead symbol(s), exactly as A does.

6. AMBIGUOUS GRAMMARS AND PARSE FOREST FILTERING

Ambiguous grammars frequently have important advantages over their unambiguous counterparts: they are shorter and simpler and result in faster parsers, smaller parse trees, and easier maintenance. Many parser generator tools, such as `bison`, permit some forms of ambiguity in conjunction with mechanisms for eliminating the resulting non-determinism in the parse table.[17] These methods include default resolution mechanisms (prefer shift, prefer earliest reduction in order of appearance in grammar) as well as a notation for expressing operator precedence and associativity [Aho et al. 1975].[18]

Resolving the conflicts in the parse table through additional information (including default mechanisms built into the parser generator) interferes with sentential-form parsing, which assumes that the parse table reflects the grammar of the language. In particular, *transitions on nonterminal lookaheads that appear to be valid may result in a parse tree that would not be produced by a batch parser*.

Figure 10 illustrates one such problem. A text edit converting addition to multiplication should trigger a re-structuring to accommodate the higher precedence of the new operator. However, the grammar is ambiguous, and a straightforward implementation of incremental sentential-form parsing produces the wrong parse tree. This situation occurs because conflict resolution encoded in the parse table is not available when the lookahead symbol is a nonterminal.

Incremental parsing methods based on state matching do not have this problem, because their incrementality derives from re-creating configurations in the pushdown automaton itself. With respect to unambiguous grammars, state matching is a sufficient but not necessary test. In the case of ambiguous grammars, however, the stronger state-matching test is useful: treating the parse table as definitive permits the incremental parser to ignore the relationship between the parse table and the grammar. State matching thus intrinsically supports any (static) conflict-resolution mechanism. Since most existing and future grammars are likely to be ambiguous, incremental sentential-form parsers will only be practical if they can also support this type of ambiguity.

One possible solution would be to encode the dynamic selection of desired parse trees in the incremental parsing algorithm itself. For example, an existing theory of operators [Aasa 1995] could be extended to produce an incremental evaluator by maintaining synthesized attributes that describe, for each expression subtree, the precedence and associativity of the 'exposed' operators within it. The incremental parser would expose operands through additional left- and right-breakdown operations in accordance with the operator specifi-

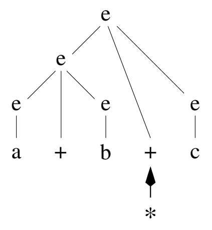
---

[17]The methods we describe in this section also apply to unambiguous grammars that are non-deterministic (with respect to a particular parsing algorithm) in the absence of conflict resolution during parse table construction.

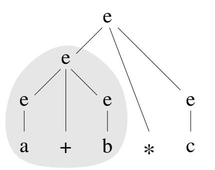
[18]These resolution methods are the most widely used, but have several theoretical disadvantages, including the fact that they may result in incomplete or even non-terminating parsers. Thorup [1994] examines methods that eliminate conflicts while preserving the completeness, termination, and performance results of classic LR parsers. Klint and Visser [1994] describe parse tree *filters*, some of which can be applied at parse table construction time.

```
%token IDENT
Order indicates precedence:
%left '+' low
%left '*' high
%%
s : e;
e : IDENT
  | e '+' e
  | e '*' e
  | '(' e ')'
  ;
```

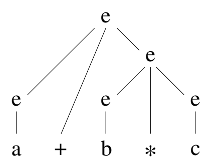
10a. Bison input file.



10b. Original program with edit site marked.



10c. Incorrect parse due to ambiguity; the reused subtree is shaded.



10d. Correct parse due to additional breakdowns.

Fig. 10. Incremental parsing in the presence of ambiguity. The grammar in (a) would be ambiguous without the precedence/associativity declarations, which control how the parser generator resolves conflicts. As shown in (c), a sentential-form parser produces the wrong parse tree, since its test for subtree reuse does not take conflict resolution into account (unlike state-matching methods). Forcing the parser to break down *fragile* productions when they occur as lookaheads will result in correctly parsed structure (d).

cations. This technique would be limited to the class of ambiguities addressed by the operator notation.

6.1 Encapsulating Ambiguity

A second, more general, solution is to employ the less efficient state-matching implementation on a restricted basis, limiting it to just those portions of the parse tree involving ambiguous constructs. In ambiguous regions, the parser uses state matching to determine the set of reusable subtrees; in unambiguous regions, sentential-form parsing can be used. State matching is required when the current state (item set) contains a conflict or when the lookahead symbol is a node constructed using state-matching (a *fragile* node). Both conditions signal the parser to switch to the more conservative subtree reuse test.

The set of fragile nodes is determined through a combination of grammar analysis and dynamic ('parse-time') tracking. First, the set of directly ambiguous productions can be output by the parser generator: these are the productions that appear in any state (item set) containing a conflict, and any node representing an instance of a production in this list is fragile. (Most parser generators already provide this information in 'verbose mode'.) The analysis applies to *any* framework that produces a deterministic parse table by selective elimination of conflicts, including both shift/reduce and reduce/reduce conflicts.[19]

Nodes can also be *indirectly* fragile; for example, a chain reduction of a fragile production would likewise be fragile. This propagation stops when a number of terminal symbols equal to the lookahead used for parsing (k) has been accumulated at the beginning and end of a fragile region; in the example of Figure 10, adding parentheses to an arithmetic expression encases the fragile region and results in an unambiguous construct.

Indirect, or *dynamic*, fragility is determined by synthesizing the exposure of conflicts along the left and right sides of a subtree, as shown in Figure 11. As each node is shifted onto parse stack its dynamic fragility can be determined: a node is explicitly fragile (and therefore requires a state) if either the left or right side of its yield exposes a fragile production. Each entry in the parse stack can be extended to include the additional information needed to track terminal yield counts and left and right conflict exposure.[20]

6.2 Implementing Limited State-Matching

Constructing a sentential-form parser that applies state matching to fragile nodes is a straightforward combination of the two algorithms. However, we prefer to avoid the additional space overhead of explicit state storage: instead of applying state matching to the portions of the parse tree not correctly handled by sentential-form parsing, these areas are simply re-created on demand. (The 'state' information on affected nodes is effectively reduced to a single boolean value.) This approach is simple, can often be implemented with no explicit storage costs whatsoever, and—given the small size of the regions affected—is very fast in practice.

In order to implement this approach, regions of the parse tree described by ambiguous portions of the grammar must be re-created whenever any modification occurs that might affect their structure. The only change required to the sentential-form algorithm is the inclusion of the boxed code in Figure 7, which replaces each fragile node appearing in the input stream with its constituents.

Unambiguous symbols (even those containing ambiguous structures, e.g., parenthesized expressions in the grammar of Figure 10) continue to be parsed as fast as before. Only a lookahead with exposed ambiguous structure must be broken down further in order to determine the next action. Fragile nodes constitute a negligible portion of the tree across a variety of programs and languages studied (C, Java, Fortran, Modula-2); the additional (re)computation has no noticeable impact on parsing performance. For grammars of practical interest, the combination of sentential-form parsing and limited state match-

[19] Visser [1995] examines an alternative approach that instead modifies the item set construction to encode parse forest filters [Heering et al. 1992]. To use this approach in conjunction with incremental parsing, the productions to which priority constraints apply must be indicated in a manner analogous to the itemization of conflict-causing productions in an LR parser generator.

[20] In practice it is unnecessary to store yield counts persistently; the count for a non-trivial subtree reused by the parser can be approximated conservatively by the minimum yield of the production it represents. If the environment already maintains the length of a subtree's text as a synthesized node attribute, this information can replace yield computation in the $k = 1$ case.

```
bool is_fragile (NODE *node) {
  return grammar→is_fragile_production(node→prog) || node→dyn_fragility;
}

class PARSE_STACK_ENTRY {
protected:
  int beginning_state;
  NODE *node;
  void push (int old_state, NODE *node);
  ...
}

Extend the normal parse stack entry object with additional fields
class EXTENDED_STACK_ENTRY : public PARSE_STACK_ENTRY {
private:
  bool left_fragile, right_fragile;
  int total_yield;
public:
  Push node onto the stack; its children are the nodes in the stack entries represented
  by the children array.
  EXTENDED_STACK_ENTRY (node, PARSE_STACK_ENTRY children[]) {
    int i;
    int num_kids = node→arity;
    Compute conservative estimate of each child's yield, as well as total yield.
    int yield[num_kids];
    for (i = 0; i < num_kids; i++) {
      if (is_token(children[i]→node)) yield[i] = 1;
      else if (has_type(EXTENDED_STACK_ENTRY, children[i]))
        yield[i] = children[i]→yield;
      else return grammar→estimate_yield(children[i]→node);
      total_yield += yield[i];
    }
    Compute and record left side's fragility.
    left_fragile = false;
    int exposed_yield = 0;
    for (i = 0; i < num_kids; i++) {
      if (grammar→is_fragile_production(children[i]→node→type) ||
          has_type(EXTENDED_STACK_ENTRY, children[i]) &&
            children[i]→left_fragile)
        {left_fragile = true; break;}
      else if ((exposed_yield = yield[i]) >= k) break;
    }
    Compute and record right side's fragility (symmetric).
    ...
    Set node's dynamic fragility status.
    node→dyn_fragility = left_fragile || right_fragile;
  }
};
```

Fig. 11. Computation of dynamic fragility.

ing uses less time and space than full state-matching parsers, while supporting the same class of conflict resolution mechanisms.

7. REPRESENTING REPETITIVE STRUCTURE

The asymptotic performance results described in this paper require the parse tree to support logarithmic search times. This is *not* the usual 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. Thus parse 'trees' are really linked lists in practice, with the concomitant performance implication: any incremental algorithms degenerate to at best linear behavior, providing 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 the sequences they construct are naturally limited; for instance, we can assume that the size of an expression in a C program is bounded in practice. The latter type are problematic, since they are substantial in any program of non-trivial length. Depending on the form of the grammar, modifying either the beginning or end of the program—both common cases—will require time linear in the length of the program text even for an optimal incremental parser.

To avoid this problem, we represent associative sequences non-deterministically; the ordering of the yield is maintained, but otherwise the internal structure is unspecified [Gafter 1990]. This convention permits the environment and its tools the freedom to impose a *balancing condition*, of the sort normally used for binary trees. (The small amount of reorganization due to re-balancing does not affect user-visible tree structure and results in a net performance gain in practice.) The appropriate data structures and algorithms are well-known [Tarjan 1983], so we will concentrate instead on the interaction of non-deterministic structure with incremental parsing.

An obvious way to indicate the freedom to choose an internal representation for associative sequences is to describe the syntax of the language using an extended context-free (regular right part) grammar [LaLonde 1977]. We can use the grammar both to specify the syntax of the language *and* to declaratively describe the representation of the resulting syntax trees. 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. Choice operators are not provided, since alternatives are conveniently expressed as alternative productions for the same grammar symbol. We will assume that any unbounded sequences are expressed in this fashion in the grammar.

Note that changes to the grammar are necessary—the parser generator cannot intuit the associativity properties of sequences, since it must treat the grammar as a declarative specification of the form of the parse tree. (Other tools will also base their understanding of the program structure on the grammar.) Associativity, while regarded as an algebraic property of the sequencing operator, is essentially a semantic notion, determined by the *interpretation* of the operators.

Since sequence specification affects only the performance of incremental parsing and not its correctness, existing grammars can be introduced to an environment and then subsequently modified to provide incremental performance. Changes required to port existing grammars (including Java and Modula-2) to our prototype environment amounted to less than 1% of their text. These changes also simplified the grammars, since regular expression notation is more compact and readable than the recursive productions it replaces.

Given a grammar containing sequence notation, we transform it to a conventional LR(k) grammar by expanding each sequence into a set of productions for a unique symbol.[21] The form of the productions expresses the associativity of the sequence; Figure 12 illustrates the transformation. (The intermediate symbol is required only for possibly-empty sequences.)

The incremental parsing algorithm requires no changes in order to process sequences.[22] The expanded grammar will be ambiguous, but—unlike conflicts in the original grammar—conflicts induced by the expansion of sequence notation do *not* require the special handling described in the previous section.

The simple 'reconstruction' approach to handling ambiguity requires that a left-recursive expansion of sequence notation result in a grammar that contains no conflicts involving the sequences themselves. Such conflicts would represent an impediment to incremental performance, requiring the sequence to be reconstructed in its entirety whenever it appeared as a lookahead symbol. The general approach to combining state-matching with the sentential-form framework does not impose this limitation, but running time increases to $O(t + s(\lg N)^2)$ without the assumption that sequences do not conflict with other productions.

7.1 Performance Analysis

Although the techniques of earlier sections always produce *correct* incremental parsers for any grammar accepted by the parser generator, the choice among grammars accepting the same language matters greatly for the sake of incremental *performance*. We now examine the assumptions that accompanied the

---

[21] Note that the most powerful transformations—those involving right-recursive expansions of sequences [Heilbrunner 1979]—cannot be employed, since the goal of non-deterministic sequences is to reuse non-trivial subtrees as they occur in the input stream, which precludes delaying all reductions until the entire sequence has been seen. The class of grammars permitted will be exactly those that are acceptable given a left-recursive expansion of all sequences. Existing techniques for constructing batch parsers *directly* from ELR(k) grammars [Sassa and Nakata 1987] cannot be used; these algorithms treat sequences in an inherently batch fashion.

[22] It is not only not necessary but *undesirable* for the incremental parser itself to restore the balancing condition. Not only would this complicate parsing, it would not assist any other transformation tool in maintaining the balancing condition. Instead, the environment should always re-balance modified sequences immediately before committing the update. Tools should perform on-line re-balancing only when performance would be severely degraded by waiting until the completion of the edit.

$$s \rightarrow L^{*}$$

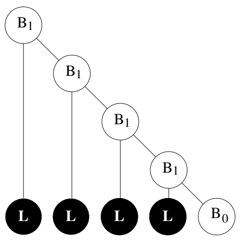
12a. Grammar with sequence notation.

$$\begin{array}{rcl} s & \rightarrow & B \\ B & \rightarrow & \epsilon \\ & | & B\ L \end{array}$$

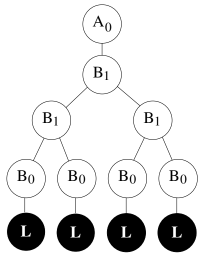
12b. Conventional left-recursive formulation.

$$\begin{array}{rcl} s & \rightarrow & A \\ A & \rightarrow & B \mid \epsilon \\ B & \rightarrow & L \\ & | & B\ B \end{array}$$

12c. Non-deterministic specification.



12d. Right-recursive tree structure.



12e. Balanced tree structure.

Fig. 12. Supporting balanced structure. Regular expressions in the grammar (a) are used to denote the associative sequences. Instead of the conventional left-linear expansion employed in batch parsing (b), each sequence operator is expanded into an additional symbol whose productions allow non-deterministic grouping (c). The tree constructed by the parser for a sequence of new tokens is initially unbalanced (d). Commit-time processing restores the balancing condition (e); the actual representation will vary depending on the exact location of modifications and the specific re-balancing algorithm used. The meta-syntax for sequence operators is summarized in Figure 13.

$$\begin{array}{rcl} \text{rhs} & \rightarrow & \text{symlist} \\ \text{symlist} & \rightarrow & \epsilon \mid \text{symlist sym} \\ \text{sym} & \rightarrow & \text{basesym} \\ & | & \text{basesym type separator} \\ & | & \texttt{(}\ \text{baselist}\ \texttt{)}\ \text{type separator} \\ \text{baselist} & \rightarrow & \text{basesym} \mid \text{baselist basesym} \\ \text{type} & \rightarrow & \star \mid + \\ \text{separator} & \rightarrow & \epsilon \mid \texttt{[}\ \text{seplist}\ \texttt{]} \\ \text{seplist} & \rightarrow & \text{basesym} \mid \text{seplist basesym} \\ \text{basesym} & \rightarrow & \textbf{ident} \mid \textbf{charlit} \mid \textbf{stringlit} \end{array}$$

Fig. 13. Meta-syntax for describing non-deterministic sequences. This is one possible notation: it differentiates between zero-or-more and one-or-more sequences, and allows multiple symbols in each sequence element as well as an optional separator. A comma-separated list of identifiers, e.g., would be written as `idlist→ID+[',']`.

performance analysis of the algorithms in Sections 4 and 5.

The basic goal is to ensure that any node in the tree can be reached in logarithmic, rather than linear, time. The tree must therefore be sufficiently well balanced; in particular, any sequence that is unbounded (in practice) must be represented as an associative sequence in the grammar. Note that non-associative sequences, though syntactically unbounded, are limited in size by semantic or pragmatic considerations. (For example, the length of individual expressions and declarations in imperative languages, rules in Prolog, and primitive forms in Lisp are all effectively bounded.)

Given the assumption that all unbounded sequences appear in the grammar using the list notation, we can only violate the performance guarantee if the interpretation of the yield of a sequence depends on its context. Consider a 'bad' grammar for the regular language $(A|B)X^+$:

$$\begin{aligned} s &\rightarrow A\ c^+ \mid B\ d^+ \\ c &\rightarrow X \\ d &\rightarrow X \end{aligned}$$

This grammar is clearly problematic, since the reduction of an X to either c or d is determined by the initial symbol in the sentence, which is arbitrarily distant. $O(|\text{sentence}|)$ recomputation is therefore needed each time the leading symbol is toggled between A and B.

Situations like this cannot arise when the interpretation of an associative sequence's terminal yield is independent of its surrounding context. In fact, as long as the contextual effect on the structure of the phrase is limited to a bounded number of terminals, the performance constraints hold. Since 'incrementalizing' a grammar to gain optimum performance already requires the determination of its associative sequences, the check for invalid dependencies can be handled by inspection.[23]

8. NODE REUSE

Incremental parsing is only one of several tools that collectively support incremental compilation and associated environment services. Overall performance is affected not just by the time it takes the incremental parser to update the program's structure, but also by the impact of the parser's changes on *other* tools in the environment. The reuse of nonterminal nodes by the parser is essential both in achieving overall environment performance and in maintaining user annotations. Figure 3 indicates the set of nodes that can be retained through explicit reuse calculation in the common case of changing an identifier spelling.

8.1 Characterizing Node Reuse

We first define and justify the concept of *reuse paths*, then discuss a specific policy for determining the set of available paths. A second, more aggressive, policy is described in the following section. Methods for computing both policies are covered in Section 8.3.

Any node reuse strategy must consider both tool and user needs. Our approach is based on a simple concept: reuse of a given node is indicated whenever its *context* or its *contents* (or both) are retained. Thus reuse is justified by exhibiting one or more paths from some base case to the node in question. Reused context typically corresponds to a path between the `UltraRoot` and a reusable node. This is referred to as *top-down* reuse. Reused content corresponds to a path from a reused token to the reusable nonterminal node, and is

[23] Fortunately, the form required for good incremental performance is also the simpler and more 'natural' expression of the syntax.

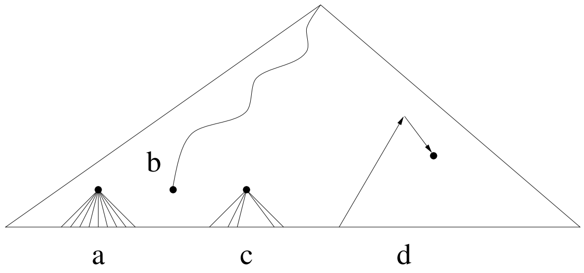


Fig. 14. Illustration of reuse paths. In each case, the circle represents a reused node, and the lines indicate the reuse path the justify its retention in the current version by linking it to some base case. (a) Unambiguous bottom-up reuse (reused contents). (b) Top-down reuse (reused context). (c) Ambiguous bottom-up reuse policy; only a subset of the children are required to remain unchanged. (d) Additional top-down reuse that can result under the ambiguous model.

referred to as *bottom-up* reuse.[24]

In both cases, the existence of such a path justifies the node's reuse by 'anchoring' it to another retained node. Given the goals of node reuse, in particular the need to avoid spurious or surprising results from the user's perspective, we also assert that the converse is true: the *absence* of such a path warrants the use of a new name for the associated nonterminal. (Note that other formulations of optimality, such as minimal edit distance, are not useful in the context of an ISDE, given the objective of preserving conceptual names for program entities.) Each reuse path establishes an inductive proof justifying the reuse of nodes along the path, in a manner that matches user intuition and is likely to improve overall environment response time. (The description of reuse paths is actually a *schema*: different policies can be employed in determining the local constraints on node reuse, generating different sets of paths in general.)

Bottom-up reuse is a natural extension of the 'implicit' node reuse that occurs when an incremental parser shifts a non-trivial subtree. In the unambiguous policy, the physical object representing a nonterminal node can be reused whenever *all* its children from the reference version are reused. Even with an optimal incremental parser, explicit bottom-up reuse checks are necessary to reverse the effect of a breakdown that turned out to be unnecessary, since an optimal choice of breakdown order cannot be known in advance (Section 5.2). Explicit reuse of modified tokens by the incremental lexer and the presence of errors in the input stream introduce additional possibilities for node reuse through explicit bottom-up checks.

Top-down reuse is defined analogously: if a node exists in both the current and previous version of the tree and its i^{th} child is changed but represents the same production in both versions of the tree, then the i^{th} child node may be reused in the new version. Figure 14 illustrates both types of reuse paths.

[24]Recall from Section 3 that the `UltraRoot` persists across changes. We also assume that the set of reused tokens is known and that the incremental lexer does not change the relative order of any reused tokens. ϵ-subtrees retained by the parsing algorithm can also serve as starting points for bottom-up reuse paths.

Several incremental parsing algorithms have tried to capture a subset of top-down reuse by implementing a *matching condition* [Larchevêque 1995; Li 1995b; Petrone 1995]. This is a test that indicates when a change can be 'spliced' into existing tree structure, thus avoiding the complete reconstruction of the spine nodes. The technique has a historical basis (it was first introduced by Ghezzi and Mandrioli [Ghezzi and Mandrioli 1980]) that has precluded better approaches: the time to test matching conditions and maintain the data needed to perform the tests outweighs the cost of a simple parsing algorithm followed by a direct reuse computation.[25]

The combination of bottom-up and top-down reuse results in *optimal* node reuse: the set of reuse paths computed are globally maximal, and no additional reuse is justified given the unambiguous policy decision. This definition of reuse is expressed without reference to a particular parsing algorithm, language, or editing model.

8.2 Ambiguous Reuse Model

The policy of restricting bottom-up reuse to only those nodes for which *all* the children are reused may appear overly restrictive. We can relax the bottom-up reuse constraint to include any case where at least one child remains unchanged. This expanded definition can only increase the total number of reused nodes, since cases of *partial overlap* with new material are now included. As an example, consider changing the conditional expression in an `if/then` statement: the statement node itself can be retained despite the replacement of one of its children.

The relaxed constraint on bottom-up reuse introduces a potential ambiguity. Consider what happens if two children of a node both exist in the new version of the tree but with different parents—which, if either, should be the reuse site? Such decisions require resolution outside the scope of syntactic reuse computation per se: the desired outcome may depend on the specific language, details of the environment, or the user's preference. The policy we adopt in our implementation is first-come/first-served; the order is determined by operational details of the parser.[26] Ambiguous bottom-up reuse can also create new starting points from which top-down reuse paths can originate (Figure 14d).

Under the ambiguous policy, the set of reuse paths is maximal (no path can be legally extended), but a global maximum is not well-defined; it depends in general on the policy for resolving 'competition' when the reuse paths do not form a tree. (However, such differences are slight, and the time required to compute more elaborate metrics—such as maximizing the total number of reused nodes—require more time to compute than they could potentially save.)

[25]In addition, the use of matching conditions precludes incremental synthesized attribution in conjunction with parsing.

[26]Other reasonable policies, such as refusing to reuse a node if there are competing reuse sites or a voting scheme based on the site with the larger number of children, are facilitated by replacing the bottom-up reuse check with the creation of a *list* of potential sites; these sites can be processed once parsing is complete and the tree is intact, but before top-down reuse takes place.

Reuse a parent when the same production is used and all children remain the same.
```
NODE *unambig_reuse_check (int prod, NODE *kids[]) {
  if (arity of prod == 0) return make_new_node(prod);
  NODE *old_parent = kids[0]→parent(previous_version);
  if (old_parent→type != prod) return make_new_node(prod);
  for (int i = 0; i < arity of prod; i++)
    if (node→is_new(kids[i])) return make_new_node(prod);
    else if (old_parent != kids[i]→parent(previous_version))
      return make_new_node(prod);
  return old_parent;
}
```

Fig. 15. Computing unambiguous bottom-up node reuse at reduction time. The reuse algorithm will either return a node from the previous version of the tree (when the production is unchanged and all the children have the same former parent) or create a new node to represent the reduction in the new tree (make_new_node). Access to the previous children is provided by the history interface presented in Figure 1.

8.3 Implementation

We now consider implementation methods for discovering bottom-up reuse during incremental parsing, and top-down reuse as a post-pass following the parse. Our methods avoid the space overhead and sub-optimal behavior associated with reuse computed through matching conditions.

Bottom-up reuse is computed most easily by adding an explicit check whenever the incremental parser performs a reduction. In the unambiguous case, each node representing a symbol in the right-hand side of the production must itself be reused and must share the same parent node from the previous version. Figure 15 illustrates this test.

Ambiguous bottom-up reuse can be computed in a similar manner by relaxing the reuse condition (Figure 16). Under this policy, only a single retained child is required to trigger the reuse of its former parent. Since the previous set of children may be split across multiple sites in the new version of the tree, this algorithm must guard against duplicate reuse of the parent by maintaining an explicit table of reused nodes during the parse. (In the unambiguous policy, competition for a single node cannot occur.)

Top-down reuse is computed as a separate post-pass. It involves a recursive traversal of the current tree, limited to the regions modified by the incremental parser. Each non-new node with one or more changed children is subject to the top-down check, which attempts to replace each new child with its counterpart from the previous version of the tree.

The algorithm in Figure 17 illustrates this process. *Reachability analysis* discovers nodes in the previous version of the tree that have been eliminated in the new version; the deleted nodes constitute the (only) candidates for top-down reuse. (Without the reachability check, top-down reuse could duplicate nodes reused implicitly by the incremental parser.) No changes to the algorithm in Figure 17 are required to support the ambiguous reuse model.

Reuse a parent when the same production is used and at least one child is unchanged.

```
NODE *ambig_reuse_check (int prod, NODE *kids[]) {
  if (arity of prod == 0) return make_new_node(prod);
  for (int i = 0; i < arity of prod; i++)
    if (!node→is_new(kids[i])) {
      NODE *old_parent = kids[i]→parent(previous_version);
      if (old_parent→type == prod && !in_reuse_list(old_parent)) {
        add_to_reuse_list(old_parent);
        return old_parent;
      }
    }
  return make_new_node(prod);
}
```

Fig. 16. Computing ambiguous bottom-up node reuse at reduction time. This method differs from that of Figure 15 by allowing a partial match to succeed: if a reuse candidate can be found among the former parents of reused children, it will be used to represent the production being reduced. A simple FCFS policy resolves competition for the same parent when its former children appear in multiple sites in the new tree. Duplicate reuse is avoided by maintaining a list of the explicitly reused nodes.

Compute top-down reuse in a single traversal of the new tree.

```
top_down_reuse () {
  compute_reachability(); Mark deleted nodes.
  top_down_reuse_traversal(UltraRoot);
}
```

Apply a localized top-down reuse check at each modification site.

```
top_down_reuse_traversal (NODE *node) {
  if (node→has_changes(local) && !node→is_new())
    reuse_isomorphic_structure(node);
  else if (node→has_changes(nested))
    foreach child of node do top_down_reuse_traversal(child);
}
```

Restore reuse paths descending from `node`.

```
reuse_isomorphic_structure (NODE *node) {
  for (int i = 0; i < node→arity; i++) {
    NODE *current_child = node→child(i);
    NODE *previous_child = node→child(i, previous_version);
    if (current_child→is_new() && !previous_child→exists() &&
        current_child→type == previous_child→type) {
      replace_with(current_child, previous_child);
      reuse_isomorphic_structure(previous_child);
    } else if (current_child→has_changes(nested))
      top_down_reuse_traversal(current_child);
  }
}
```

Fig. 17. Computing top-down reuse. The algorithm performs a top-down traversal of the structure that includes each modification site, attempting to replace newly created nodes with discarded nodes. `compute_reachability` identifies the set of nodes from the previous version of the tree that were discarded in producing the current version; these nodes are the (only) candidates for top-down reuse.

8.4 Correctness and Performance

Adding explicit reuse to the incremental parser can never result in a node being used twice. Unambiguous bottom-up reuse avoids node duplication by construction. Bottom-up reuse in the ambiguous model and top-down reuse both contain an explicit guard against duplication. Each bottom-up check is performed in constant time and adds no significant overhead to incremental parsing. Top-down reuse does not affect the asymptotic results in Section 7, since only nodes touched by the incremental parser are examined. The combination of optimistic sentential-form parsing, reuse checks at reduction time, and a separate top-down reuse pass results in optimal reuse in the unambiguous case and a maximal solution in the ambiguous model, computed in optimal space and time.

Our preferred approach in practice is to apply ambiguous bottom-up reuse *without* the top-down pass: this locates virtually all the reusable nodes *including* most of those on top-down reuse paths. (Only some cases involving ϵ-subtrees and chain rules can be missed, when no children exist to anchor the parent node's reuse.) In the example shown in Figure 3, this simple method results in only one changed node in the entire tree: the modified token.[27]

9. CONCLUSION

This paper provides four main research contributions. First, it offers a general algorithm for incremental parsing of LR grammars that is optimal in both time and space and supports an unrestricted editing model. Existing techniques for constructing LR(k), LALR(k), and SLR(k) parsers can be used with very little modification.

Second, it extends sentential-form parsing theory to permit the use of ambiguous grammars (in conjunction with static disambiguation mechanisms), allowing the sentential-form approach to apply to grammars in widespread use. Extensions to the parsing algorithm to support static filtering of the parse forest are both simple and efficient.

Third, it describes the importance of balancing lengthy sequences, providing a solution in terms of grammar notation, parse table construction, and run-time services. In conjunction with this representation, a realistic performance model is offered that allows for meaningful comparisons with batch parsing and other incremental algorithms.

Finally, we define optimal node reuse independent of the operational details of parsing. General models of ambiguous and unambiguous reuse are presented, along with simple and efficient methods to implement both approaches.

ACKNOWLEDGMENTS

We are grateful to the *Ensemble* developers for helping to create an appropriate testbed for this project. Special thanks go to John Boyland, Todd Feldman,

[27]If the lexer reuses this token, the tree will possess no changes whatsoever after the lexing/parsing analysis. Obviously the user's modification has *semantic* significance; the original edit, along with its path information, remains available to tools, such as semantic analysis, for their own analyses.

William Maddox, Vance Maverick, and the anonymous reviewers for their comments on drafts of this paper.

REFERENCES

AASA, A. 1995. Precedences in specifications and implementations of programming languages. *Theor. Comput. Sci. 142,* 1 (May), 3–26.

AGRAWAL, R. AND DETRO, K. D. 1983. An efficient incremental LR parser for grammars with epsilon productions. *Acta Inf. 19*, 369–376.

AHO, A. V., JOHNSON, S. C., AND ULLMAN, J. D. 1975. Deterministic parsing of ambiguous grammars. *Commun. ACM 18,* 8 (Aug.), 441–452.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass.

BALLANCE, R. A., BUTCHER, J., AND GRAHAM, S. L. 1988. Grammatical abstraction and incremental syntax analysis in a language-based editor. In *Proceedings of the ACM SIGPLAN '88 Symposium on Compiler Construction*. ACM Press, Atlanta, Ga., 185–198.

BEETEM, J. F. AND BEETEM, A. F. 1991. Incremental scanning and parsing with Galaxy. *IEEE Trans. Softw. Eng. 17,* 7 (Jul.), 641–651.

BURKE, M. G. AND FISHER, G. A. 1987. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Trans. Program. Lang. Syst. 9,* 2 (Apr.), 164–197.

CORBETT, R. 1992. `bison` release 1.24.

DEGANO, P., MANNUCCI, S., AND MOJANA, B. 1988. Efficient incremental LR parsing for syntax-directed editors. *ACM Trans. Program. Lang. Syst. 10,* 3 (Jul.), 345–373.

GAFTER, N. M. 1990. Parallel incremental compilation. Ph.D. dissertation, University of Rochester, Rochester, N.Y.

GHEZZI, C. AND MANDRIOLI, D. 1980. Augmenting parsers to support incrementality. *Journal of the ACM 27,* 3 (Jul.), 564–579.

HEDIN, G. 1992. Incremental semantic analysis. Ph.D. dissertation, Department of Computer Science, Lund University.

HEERING, J., HENDRIKS, P. R. H., KLINT, P., AND REKERS, J. 1992. *The syntax definition formalism SDF — Reference Manual.* ASF+SDF Project.

HEILBRUNNER, S. 1979. On the definition of elr(k) and ell(k) grammars. *Acta Inf. 11*, 169–176.

JALILI, F. AND GALLIER, J. H. 1982. Building friendly parsers. In *9th ACM Symp. Principles of Prog. Lang.* ACM Press, Albuquerque, N.Mex., 196–206.

KLINT, P. AND VISSER, E. 1994. Using filters for the disambiguation of context-free grammars. In *Proc. ASMICS Workshop on Parsing Theory*. Milan, Italy.

LALONDE, W. R. 1977. Regular right part grammars and their parsers. *Commun. ACM 20,* 10, 731–740.

LARCHEVÊQUE, J. M. 1995. Optimal incremental parsing. *ACM Trans. Program. Lang. Syst. 17,* 1, 1–15.

LI, W. X. 1995a. A simple and efficient incremental LL(1) parsing [sic]. In *SOFSEM '95: Theory and Practice of Informatics* (Milovy, Czech Republic). LNCS. Springer-Verlag, Berlin, 399–404.

LI, W. X. 1995b. Towards generating practical language-based editing systems. Ph.D. dissertation, University of Western Australia.

MADDOX, W. 1997. Incremental static semantic analysis. Ph.D. dissertation, University of California, Berkeley. Available as technical report UCB/CSD–97–948.

MAVERICK, V. 1997. Presentation by tree transformation. Ph.D. dissertation, University of California, Berkeley. Available as technical report UCB/CSD–97–947.

MURCHING, A. M., PRASAD, Y. V., AND SRIKANT, Y. N. 1990. Incremental recursive descent parsing. *Computer Languages 15,* 4, 193–204.

PETRONE, L. 1995. Reusing batch parsers as incremental parsers. In *Proc. 15th Conf. Foundations Softw. Tech. and Theor. Comput. Sci.* (Bangalore, India). Number 1026 in LNCS. Springer-Verlag, Berlin, 111–123.

REPS, T. W. AND TEITELBAUM, T. 1989. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, Berlin.

SASSA, M. AND NAKATA, I. 1987. A simple realization of LR-parsers for regular right part grammars. *Information Processing Letters 24*, 113–120.

SHILLING, J. J. 1992. Incremental LL(1) parsing in language-based editors. *IEEE Trans. Softw. Eng. 19,* 9 (Sep.), 935–940.

TARJAN, R. E. 1983. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, Penn.

THORUP, M. 1994. Controlled grammatic ambiguity. *ACM Trans. Program. Lang. Syst. 16,* 3 (May), 1024–1050.

VAN DE VANTER, M. L., GRAHAM, S. L., AND BALLANCE, R. A. 1992. Coherent user interfaces for language-based editing systems. *Intl. J. Man-Machine Studies 37*, 431–466.

VISSER, E. 1995. A case study in optimizing parsing schemata by disambiguation filters. Tech. Rep. P9507, Programming Research Group, University of Amsterdam. Jul.

WAGNER, T. A. 1997. Practical algorithms for incremental software development environments. Ph.D. dissertation, University of California, Berkeley. Available as technical report UCB/CSD–97–946.

WAGNER, T. A. AND GRAHAM, S. L. 1997a. Efficient self-versioning documents. In *CompCon '97*. IEEE Computer Society Press, San Jose, Calif., 62–67.

WAGNER, T. A. AND GRAHAM, S. L. 1997b. History-sensitive error recovery. In preparation.

WEGMAN, M. N. 1980. Parsing for structural editors. In *Proc. of 21st Annual IEEE Symposium on Foundations of Computer Science*. IEEE Press, Syracuse, N.Y., 320–327.

YANG, W. 1994. Incremental LR parsing. In *1994 International Computer Symposium Conference Proceedings*. National Chiao Tung University, Hsinchu, Taiwan, 577–583 vol. 1.

YEH, D. AND KASTENS, U. 1988. Automatic construction of incremental LR(1) parsers. *ACM SIGPLAN Notices 23,* 3 (Mar.), 33–42.