Practical Algorithms for Incremental Software Development Environments

Tim A. Wagner

March 10, 1998

Preface

We describe an integrated collection of algorithms and data structures to serve as the basis for a practical incremental software development environment. A self-versioning representation provides a uniform model that embraces both natural and programming language documents in a single, consistent framework. Software artifacts in this representation provide fine-grained change reports to all tools in the environment. We then present algorithms for the initial construction and subsequent maintenance of persistent, structured documents that support unrestricted user editing. These algorithms possess several novel aspects: they are more general than previous approaches, address issues of practical importance, including scalability and information preservation, and are optimal in both space and time. Since deterministic parsing is too restrictive a model to describe some common programming languages, we also investigate support for multiple structural interpretations: incremental non-deterministic parsing is used to construct a compact form that efficiently encodes syntactic ambiguity. Later analyses may resolve ambiguous phrases through syntactic or semantic disambiguation. This result provides the first known method for handling C, C++, COBOL, and FORTRAN in an incremental framework derived from formal specifications.

Our transformation and analysis algorithms are designed to avoid spurious changes, which result in lost information and unnecessary recomputation by later stages. We provide the first non-operational definition of optimal node reuse in the context of incremental parsing, and present optimal algorithms for retaining tokens and nodes during incremental lexing and parsing. We also exploit the tight integration between versioning and incremental analysis to provide a novel *history-sensitive* approach to error handling. Our error recovery mechanism reports problems in terms of the user's own changes in a language-independent, non-correcting, automated, and fully incremental manner.

This work can be read at several levels: as a refinement and extension of previous results to address issues of scalability, end-to-end performance, generality, and description reuse; as a 'cookbook' for constructing the framework of a practical incremental environment into which semantic analysis, code generation, presentation, and other services can be plugged; and as a set of separate (but interoperable) solutions to open problems in the analysis and representation of software artifacts. Our results are promising: in addition to embracing a realistic language model, both asymptotic and empirical measurements demonstrate that we have overcome barriers in performance and scalability. Incremental methods can now be applied to commercially important languages, and may finally become the standard approach to constructing language-based tools and services.

Acknowledgments

Many thanks to the members of the *Ensemble* research group, especially John Boyland, William Maddox, and Vance Maverick, for their assistance. John Boyland provided the `program` style and `makebib` tools for typesetting this report. Marsha Lovett provided the cognitive psychology results cited in Chapter 7. Emory Bunn and Chris Genovese advised me on math and statistics questions. The comments and suggestions of my dissertation committee—Professors Susan Graham, Michael Harrison, and Philip Stark—were essential in improving the quality of this dissertation. Any errors, omissions, or inconsistencies that remain are entirely the fault of the author.

This research has been sponsored in part by the Advanced Research Projects Agency (DARPA) under Grant MDA972-92-J-1028. The content of this work does not necessarily reflect the position or policy of the U. S. Government.

Copyright Information

Tim A. Wagner
573 Soda Hall, #1776 LeRoy
Computer Science Division
EECS Department
University of California
Berkeley, CA 94720
`twagner@cs.berkeley.edu`
`http://http.cs.berkeley.edu/~twagner/`

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Delays in developing, maintaining, and understanding software documents continue to affect the time to market for software companies and the time to deployment within corporate MIS divisions. Despite advances in many areas—computer and network hardware, programming languages, and user interfaces—the compilation and analysis of programs remains a bottleneck in the development process and a limiting factor in programmer productivity. Increasingly sophisticated languages, software systems, and analysis demands have outpaced aggressive performance growth in both computing hardware and network bandwidth, resulting in a net *increase* in compilation delays [2]. In this chapter we discuss methods to mitigate these problems, then motivate and characterize our solution in the form of software development environments that operate *incrementally*.

1.1 Software Development Environments

The term 'environment' is typically used to describe an integrated collection of tools that assist the programmer in developing or maintaining software artifacts [12, 29, 47, 83]. Though historically derived from a collection of independent programs—compilers, debuggers, recompilation managers, etc.—the software development environment (SDE) attempts to be more than the sum of its parts. Its twin goals are to simplify and speed the development process through a tighter integration of the underlying tools. Benefits can be realized through common user interface conventions, control integration, and low-bandwidth data sharing among the tools. The implementation also benefits from functionality specialization: by working cooperatively, tools can avoid duplication of effort in many cases.

The first goal, simplifying the development process, is well addressed by the state of the practice in commercial SDEs: interactive design and better tool cooperation, along with such useful additions as hypertext manuals, have clearly reduced the learning curve for these complex tools [15, 68]. Unfortunately, the second goal—improved productivity—has gone unrealized. Since these environments can operate no faster than the collection of batch programs they replace, development time for an experienced programmer—and overall productivity—remain essentially unchanged.

The limitations of existing systems are a direct result of their continued reliance on batch technologies. The need to reconstitute the syntax and semantics of a file or module 'from scratch' whenever a small change is made (either directly or in some related module) forces the programmer to wait after virtually every update. Increased complexity in languages and improved compile-time checking, coupled with the growing size of software systems have outpaced gains in hardware and network performance: recompilation after a minor change to the program often takes minutes or even hours to complete [2]. Lengthy delays decrease productivity and cause programmers to lose track of their working context when attempting to debug or maintain a complex program [93]. Despite the appearance of an array of techniques to salvage incrementality without discarding batch algorithms (parallel `make`, incremental linking, pre-compiled system headers), compilation delays remain, and remain fundamentally limiting.

Furthermore, the batch nature of these environments precludes an entire class of useful services that can be provided only in the presence of a persistent structural representation and incremental consistency maintenance. One example is error recovery: because each compilation unit is recomputed in its entirety whenever the user modifies it (directly or indirectly), error recovery is limited to guessing strategies based on the state of the current analysis. In our approach to error recovery, discussed in Chapter 8, the user's *own* changes are the basis of error handling and presentation. This represents a new service paradigm: user-centered tools that understand the history of the document and provide real-time feedback. The combination of incrementality and the availability of a fine-grained development log enables the construction of powerful interactive services that are impossible (or impractical) to provide in a batch environment.

Other services, encumbered by the batch nature of the underlying implementation, sacrifice correct semantics to achieve reasonable performance. For instance, program databases can provide the developer with a rich set of data to assist in browsing, understanding, and transforming programs. However, changes to the program are often not immediately reflected in the content of the database, allowing it to produce incorrect results. Restoring database consistency is so time-consuming that it is often done overnight [17].

Incremental software development environments (ISDEs) provide the foundation to maintain a consistent executable image, database, or other program-wide analysis results with truly interactive speeds [6, 8, 54, 77, 66, 81]. By integrating the (typically small) set of new changes with the results of a previous analysis, incremental services can operate 'instantaneously'. At the same time, the persistent representations used by these algorithms allow high-bandwidth data integration that enables new classes of functionality: services like history-sensitive error recovery are not only possible, but actually *easier* to implement than the conventional batch alternatives.

However, despite their promise, and more than two decades of research and investment, incremental environments have not enjoyed either commercial success or widespread use. The reasons are simple: existing systems have failed to address key issues in generality, efficiency, and scalability.

Many early approaches to building incremental systems imposed restricted editing models that were simply unrealistic—programmers cannot be asked to give up the power and flexibility of arbitrary text editing to simplify the implementation of the environment. Even worse, the *language model* proposed by previous systems has been too limited to address the needs of many real programming languages, including C, C++, COBOL, and FORTRAN; even such 'academic' languages as Haskell, Icon, and Oberon possess features that are difficult, if not impossible, to model within the framework of existing incremental systems. Prototypes that addressed the needs of toy languages and commercially irrelevant languages, such as Pascal, had no obvious migration path to handling more challenging (and useful) languages.

Despite its centrality, the incrementality of the algorithms used in previous systems was often imperfect.[1] Most published methods for incremental parsing, for example, exhibit running times that are *linear* in the size of the entire file or module, even in some common case editing scenarios. Many incremental algorithms and environments ignored both constant factors and scalability, with the result that large programs could not be represented successfully. Research ISDEs have also ignored existing specification formats in favor of proprietary formalisms, imposing needless burdens on the specification writer. Commercial adaptation of incremental technology, though it offers an overwhelming potential for decreasing the time and cost of producing software products, cannot be expected to occur until the research community provides convincing demonstrations using the languages and large-scale programs that reflect real-world development.

Our goal is to address these needs through the design of an effective, multilingual ISDE, providing interactive services for the development, maintenance, and documentation of complex software systems. By avoiding delays and preserving the user's working context, overall productivity can be significantly enhanced. By providing tighter tool integration, especially with respect to the fine-grained history log, existing services can be improved and new classes of functionality can be provided that are impractical in a batch setting.

1.2 Requirements

A successful ISDE must exhibit a number of characteristics:

Support for multiple languages
The environment must be multilingual, supporting multiple documents in different languages, and single (compound) documents expressed in multiple languages.

Mostly declarative language description
Language-specific analysis and transformation tools should be automatically generated from high-level, declarative descriptions. Compiled tools should be dynamically loaded into the running environment as needed. (This allows new languages and services to be introduced without shutting down the environment.) It should be possible to merge tools generated from formal specifications with others described procedurally, in order to provide flexibility, backward compatibility, and a simple integration mechanism for prototyping new services.

General language model
The language model should be (at least) sufficient to embrace C, C++, COBOL, FORTRAN, Java, Lisp, and the common dialects and versions of these languages.

Uniform, structured document representation
Software artifacts and natural language documents should be represented in the same fashion. Common editing, navigation, storage, and presentation services can then be employed across languages and document types in addition to allowing for customization on a per-language or per-document basis.

[1] Beyond their obvious restriction to single-language development, this appears to have been the central technological problem with commercial monolingual environments, such as SmartSystem™ [77].

Unrestricted editing model
The environment should not contain inherent restrictions on the type of edits (textual, structural), their location within the program, or the timing of user modifications with respect to consistency restoration.

Efficiency
Constant factors in both time and space consumption should be reasonable with respect to hardware capabilities *and* projected increases in the size and complexity of the software systems being developed. Efficiency should be demonstrated by empirical measurement of common operations on real programs.

Scalability
The needs of large source files, modules, and programs must be effectively addressed. Analysis or transformation algorithms that require time or space linear in the size of the document, rather than the changed portion, are inherently unscalable and therefore inappropriate in the implementation. Where possible, scalability should be ensured by an asymptotic analysis based on realistic assumptions.

1.3 Scope of Work

Our work encompasses the definition of a persistent (versioned) structural document representation and algorithms for maintaining the consistency between its structure and content, including the handling of errors. These topics must be considered as a whole if the results are to be interoperable, efficient, and scalable. (Chapter 8 expands on the need to treat these subjects in conjunction.) Other services, including semantic analysis and presentation concerns, can be considered separately. While their functionality is related to, and partially dependent on, the representations developed here, their implementations are distinct. The primary interaction with additional tools and services is through the change reporting and document navigation interfaces described in Chapters 3 and 4.

1.4 Outline of the Report

In Chapter 2, we continue the introduction by describing the *Ensemble* environment, the research prototype in which our work was performed. *Ensemble's* goals, background, architecture, and implementation are discussed.

Chapter 3 presents a model of *self-versioning documents*, documents that can store their own history efficiently and produce incremental change reports for clients. Low-level versioned objects are constructed using conventional object-oriented techniques, then combined into a structured framework to provide versioned documents. We correct previous theoretical work on persistent graphs and extend these results to provide an efficient mechanism for document representation in an ISDE.

Chapter 4 builds on the results of the previous chapter, discussing models for editing, analyzing, and transforming programs and other formal language documents. This chapter also covers the interaction between analysis/transformation algorithms and versioning/history services, including a discussion of information preservation through *node reuse* techniques during transformations. The run-time language object model, which represents instances of language-specific analysis and transformation services, is also described here.

Chapter 5 introduces the first of several analysis algorithms. We begin with a novel approach to incremental lexing that possesses features lacking in existing systems. Existing (batch) lexical specification formalisms are permitted in all their generality: unbounded lookahead, user-controlled start states, and match rejection; in addition, we admit a clean integration with *ad hoc* procedural pattern recognition code. The incremental lexical analyzer we develop has novel support for token reuse and unrestricted editing as well as optimal time and space results.

In Chapter 6 we present the first of two incremental parsing methodologies. *Sentential-form parsing*, covered in this chapter, is designed for deterministic languages. It is the simplest and most general parsing technique for common grammar classes (LR(1), LALR(1)), runs in optimal time and space, and supports both textual and structural editing. Support for grammatical ambiguity, balanced sequences, and node reuse during parsing is also covered in this chapter.

Chapter 7 presents a second approach to incremental parsing based on *non-deterministic* methods. These techniques allow the construction of an intermediate form capable of directly expressing unresolved ambiguities in the context-free syntax of a program. Further analysis (including incremental static semantic analysis) can continue the resolution process through a variety of dynamic disambiguation techniques. This approach provides a truly general language model, allowing the natural context-free syntax of C, C++, Fortran and other languages to be expressed directly.

Chapter 8 combines incremental analysis with fine-grained versioning to provide *history-sensitive error reporting*. Information about the program's history enables a non-correcting recovery and a presentation of errors in terms of the user's own modification sequence. This method is automatic, language-independent, and provides more informative recoveries that any previous batch or incremental system.

Appendix A contains two algorithms required by the self-versioning document representation developed in Chapter 3: changing the cached version of an object when its history is recorded differentially and computing nested changes during a specified time period.

Appendix B contains the algorithm for incremental non-deterministic parsing developed in Chapter 7.

Appendix C discusses the integration of explicit whitespace, embedded comments, and other 'non-grammatical' material with the persistent program representation. Two mechanisms are provided for creating and maintaining the additional structure: a transformation of the original grammar and an extension to the incremental parsing algorithm.

Chapter 2

The *Ensemble* Environment

Ensemble is an interactive Unix program for displaying, editing, analyzing, and transforming both software artifacts and natural language documents. Users interact with this system to view, create, modify, maintain, and understand collections of documents. Figure 2.1 shows a screen dump of the running environment.

The *Ensemble* research project centers around this prototype implementation. *Ensemble* marries two lines of research: the development and maintenance of information-rich software artifacts and the authoring and presentation of multimedia natural language documents. Primary areas of investigation include designing a uniform model of documents, achieving true interactivity through the use of incremental algorithms and appropriate representations, and establishing a coherent dialog with the user through multiple presentations, unrestricted editing, and extensive customization. Our research provides the basis for some of this functionality and builds on the results of others' efforts.

2.1 Background

The *Ensemble* research project was begun in 1990 at the University of California at Berkeley by Professors Susan L. Graham and Michael A. Harrison. Over 20 graduate and undergraduate students have contributed to its implementation.

As a research program, *Ensemble* represents the confluence of two earlier projects: *Pan* [8], a prototype of an incremental, multilingual programming environment, and VORTEX [16], an interactive document typesetting system formulated as an incremental version of TEX [55]. The synthesis of these two lines of research provided leverage in the form of local expertise, common environment functionality, and interesting synergism between algorithms for incremental analysis and transformation developed in the context of programming language environments, and powerful mechanisms for typesetting, presentation, and user interaction formulated for natural language documents.

While both projects had independent research goals that continued in the context of the joint system, the integration raised additional research questions: could a uniform model of programming and natural language documents be developed, and would the tight integration of presentation and analysis/transformation services provide significant benefits in terms of user services, implementation leverage, or design simplicity? (We discuss the answers in Section 2.5.)

2.2 Design Principles

Several design principles have served as long-term goals, guiding the direction of research in *Ensemble* and determining choices in its implementation. Here we survey the most important criteria.

- Incrementality is the basis of powerful interactive services. Incrementality is not limited to editing or presentation; it includes everything conventionally described as 'compilation'.
- Tight integration (control *and* data) between tools is a necessary design strategy for interactive environments, and encourages new types of functionality lacking in current systems.
- A common document model must embrace both programs and natural language documents. There should be a single way of storing, viewing, editing, navigating, and reporting changes in a document. The document model must include multimedia elements, compound documents, multiple interpretations of structure, and multiple-language documents. Document 'content' should be distributable; information about a document may be expressed as structure, media-specific data, node attributes, annotations, or database entries.

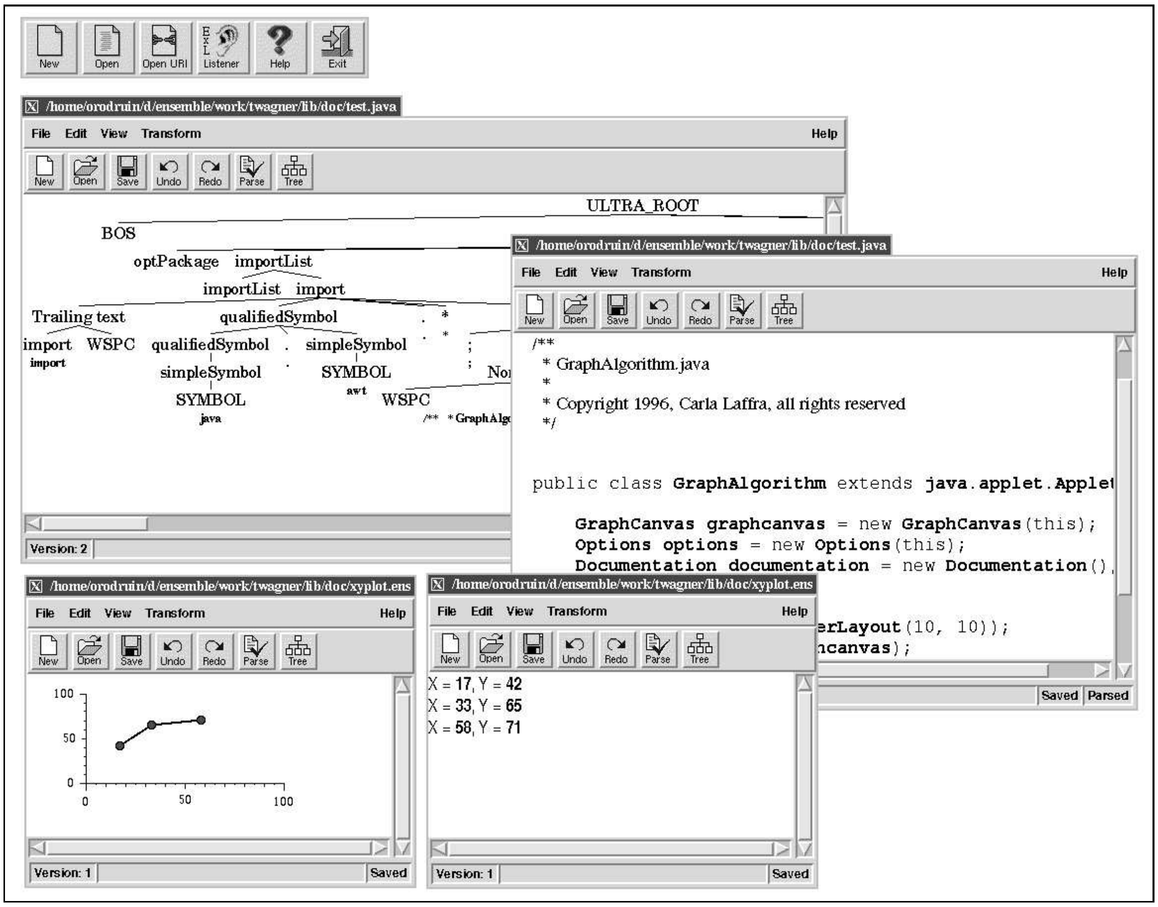


Figure 2.1: Screendump of *Ensemble*. Two documents have been loaded: a Java program and a dataset. Both structural and textual presentations of the program are shown. The dataset is displayed as a graph on the left and as text on the right.

- Generality in document manipulation is crucial to acceptance; this implies an unrestricted editing model, but also requires multiple presentations and multiple views, customizable interaction, and both language-independent *and* language-specific behavior.
- Multiple languages must be supported. Language descriptions should be as high-level as possible. The language model must be realistic, capturing the languages of significant commercial and academic interest.
- Efficiency, scalability, and usability issues must be part of the design; they cannot be retro-fitted.
- Customization and extendibility are necessary at every level. Multiple mechanisms for achieving these goals must be present to provide a variety of points in the power/performance/ease-of-use design space.

2.3 Architecture

Ensemble as a system is divided into two conceptual entities: a *language specification system* and a *run-time environment*. The latter is the environment proper: it includes the services to edit, analyze, and transform structured documents. Some of this functionality is provided in a language-neutral fashion; other services must be specialized for the language in which a document is written. Code and data specific to a particular language are intentionally separate from the core environment.

2.3.1 Language Specification

Ensemble's language-specific elements are typically derived from formal specifications. Each language description is translated into a set of C++ files, which are then compiled and linked to produce a shared ('dynamically linked') library. This compilation is performed off-line, permitting optimizations that result in time- and space-efficient representations and algorithms. Multiple dialects and multiple versions of each language can be supported simultaneously in the environment.

Several language-specific services are currently supported. These include incremental lexical, syntactic, and semantic analyses; structure-based transformations; and specification-based presentation. The language model is object-based: these services are produced by sub-classing, adding language-specific code and/or data to parameterize a generic algorithm. A separate class is used to represent the language as a whole, providing run-time access to its grammar, instances of its analysis and transformation tools, and any additional information or services particular to the language.

The shared libraries representing compiled language objects are loaded on demand into the running environment. This arrangement makes it possible to support a large number of languages without the environment itself becoming unmanageably large, and precludes the need to specify the set of available languages when the environment is built.[1] Compound documents are supported in a natural manner by using several language objects simultaneously.

Language specification *per se* is not part of the research objective of this dissertation. Our specification methods intentionally reuse the notation of familiar batch tools whenever possible, in order to decrease the time and effort required to port existing language descriptions to *Ensemble*. Other *Ensemble* researchers have investigated specification language design in the context of particular domains, including semantic attribution [63] and presentation [65, 70].

2.3.2 Document Model

Documents in *Ensemble* are always represented *structurally*, although users typically choose to view and edit them as text. Each node in the document structure is an instance of a C++ class identified with a production in the language's grammar. Information specific to a particular production is translated into data fields in the appropriate class; routines to compute these attributes become methods. Attributes that sparsely populate the document structure can be represented as annotations ('property lists') or as separate maps. Each terminal symbol in the grammar is translated into a token class.[2] In this report we concentrate primarily on programs and other documents expressed in formal languages. Discussions of representations, history services, and editing models should be understood to apply to *all* document types.

In a program, lexical analysis partitions the text into the lexemes of the tokens; parsing produces the structure of the program by creating nodes and edges to represent the parse tree. Semantic analysis attributes the nodes with cached attribute values, and updates maintained relations in the program database. Because the environment is interactive, these operations are achieved *incrementally* by repairing only the regions of the program affected by the user's modifications since the previous analysis. A new program can be introduced to the environment by treating it as a single text insertion into an (otherwise empty) structural representation.

[1] Thus *Ensemble* is not a 'template-based' monolingual environment as are the environments produced by the Synthesizer Generator [81].

[2] In addition to text, terminals and node attributes can consist of such multimedia elements as graphic objects, images, audio/video clips, etc. The representation and editing of multimedia components and natural language documents is described elsewhere [65]. Additional tokens are used to represent material outside the grammar, including explicit whitespace, textual comments, and so forth.

Incrementality is only productive when it results in rapid response times for the user. This constraint on performance requires care in the choice of a document representation: to provide rapid access to every section of the document, lengthy sequences must be represented in a balanced fashion. A balanced representation of sequences guarantees logarithmic access time to each node, even for 'unstructured' documents such as simple text. Subsequent chapters explore the specification and representation issues surrounding sequences in greater detail.

Ensemble supports a fully general editing model. Text editing can change the contents of the tokens by inserting, deleting, or overwriting characters; structural editing can alter the shape of the tree. It is not necessary for either type of editing to maintain a legal representation with respect to the language definition: the user will eventually invoke the analysis services to restore consistency to the program. There is no restriction on the location or types of edits or on the timing of consistency restoration with respect to modifications.

All *Ensemble* documents are *persistent*: each direct modification introduces a conceptually distinct version. Any changes needed to restore consistency when the user requests re-analysis are also treated as an (atomic) update. Unlike existing commercial or research systems, *Ensemble* can restore any version quickly, even if it involves reversing or re-applying a complex, language-based transformation.

We focus primarily on single-language programs represented as trees. *Ensemble's* compound document architecture has been described elsewhere [65], and is largely orthogonal to the issues discussed here. Chapter 7.3.2 investigates the construction of multiple representations within a single program module through non-deterministic parsing.

2.3.3 Presentations and Views

Several presentation services have been developed within the context of *Ensemble*, providing specification-driven display based on the document content model. These include general purpose tools, such as Proteus [39, 70] and tree-transformation systems [40], as well as several specialized tools for program presentation [65]. Each presentation instance is derived from a *presentation schema*, separate from the document content, which determines the appearance of the document. In addition, the user can override (temporarily or permanently) any automatically generated presentation attribute.

Multiple presentations, using the same or different schemas, may be active simultaneously. Multiple presentations are useful for natural language documents; for example, the user can compare a one-column presentation to a two-column version. However, it is also a powerful tool for formal documents, allowing the structure (in the form of a tree) to be viewed next to a conventional text-based display.[3] In keeping with the unrestricted editing model, *all* presentations of document content are editable.

2.3.4 Customization and Extensibility

Ensemble supports customization and extension at a variety of levels. From the point of view of this work, language-specific functionality represents the most important customization of the core environment. New languages can be described, compiled, and dynamically loaded to extend or modify (through replacement of an existing language object) a running environment. Dynamic loading can be used to add new environment services in other areas, facilitating rapid prototyping of the environment itself.

Central elements of *Ensemble's* domain model are exported through ExL, a Lisp-based extension language [24]. As in the EMACS editor, key and mouse bindings can be modified through ExL, which also provides customization of cursor management and event mapping in compound documents.

Ensemble's user interface is written in Tcl/Tk [74], making it easy to add new features or to change existing elements of the user interface quickly. The user interface is accessible from ExL, allowing extension writers to associate visual components with new functionality.

New presentations can be created and used to alter the display of a document. Users can override the default attributes assigned by a specific presentation, and can choose to make the settings permanent.

The user can select optional analyses to apply to any loaded document. Additional semantic descriptions can be introduced to the environment; these descriptions can make use of existing analyses, as well as exposing their results through a program database, the user interface, or the extension language.

Transformations can be applied to documents from a suite of existing transformation services. New transformations can be added as part of an existing language specification or as stand-alone tools.

[3] Multiple *views* of a single presentation are also provided. Views encapsulate device-dependent details by rendering a presentation onto a specific physical device.

2.4 Implementation

The environment itself runs as a single-threaded Unix process on a variety of platforms, including SunOS, Solaris, HPUX, and Linux. The user interface runs as a separate process, connected via a local socket. *Ensemble* currently consists of approximately 300,000 lines of code, most of it in C++. (The remainder is written in C, ExL, and Tcl/Tk.)

Language specifications exist for C, Exl, Fortran, Java, and Modula-2, though not all languages possess semantic descriptions. Lexical descriptions are written using the `flex` specification language [75]. Grammars can be written in either extended BNF or the syntax of `bison` [18]. Semantic attribution is defined by a specialized language, `adl` [63].

The algorithms described in subsequent chapters have all been implemented as part of *Ensemble*, and have been tested with multiple languages and multiple documents. All measurements were conducted using `gprof` or `quantify`™ on an otherwise-unloaded sparc20™ processor.

2.5 Retrospective and Future Work

The original question posed by the integration of *Pan* and VORTEX has been answered: a common document model is not only possible, but desirable as a design principle *and* as an implementation strategy. Tight integration between document processing technology and language-based analysis and transformation mechanisms has provided the *Ensemble* architecture with powerful capabilities lacking in both commercial and research platforms.

However, *Ensemble* remains strictly a prototype; further development is required before this experiment can be considered complete. Shortcomings in the current implementation limit the ability to customize the editing model to accommodate differences between formal and natural language documents, particularly with respect to explicit whitespace and comments (Appendix C). While coarse-grained compound documents are already available, extensive presentation and semantic analysis support for multiple structural interpretations (Chapter 7) is incomplete.

Current research efforts involve the construction of additional correctness-preserving transformations (including code generation by tree transformation), support for non-deterministic semantic attribution, and the extension of the user interface to provide additional access to document history and change information. *Ensemble's* storage model will be replaced in the future with an object-oriented database. Other 'programming-in-the-large' features are needed, including support for macro languages and a richer model of the translation process itself. Semantic analysis is being extended to support on-the-fly queries of the program database. Existing language specifications are being revised in accordance with emerging standards.

Part I

Versioning and History Services

Chapters 3 and 4 describe the representation of software artifacts in *Ensemble*. Special attention is given to the problems of fine-grained versioning and *change reporting*—the mechanisms by which client services record and communicate updates. Chapter 3 describes the low-level representation itself, providing a uniform model for creating *self-versioning documents*. In Chapter 4 we focus on one particular document type, programs, which will be of central importance to later chapters. This chapter develops a model for editing and transforming programs and discusses the use of versioning and history services by the algorithms that maintain consistency between the program's content and structure. Part II of this report focuses on the incremental analysis algorithms themselves.

Chapter 3

Efficient Self-Versioning Documents

This chapter discusses methods for producing software and multimedia documents that are *self-versioning*—they efficiently capture changes as the document is modified, providing access to every version with extremely fine granularity. The approach uses an object-based spatial indexing scheme that combines fast access with very low storage overhead. Multiple tools can extract change reports from these documents without requiring their queries to be synchronized. We describe and evaluate a working implementation of these ideas, suitable for use in software development environments, multimedia authoring systems, and non-traditional databases.

3.1 Introduction

Documents based on linked, hierarchical data structures with media content in their leaves are at the core of software development environments, multimedia authoring systems, and various types of non-traditional ('engineering') database implementations [52]. The management of versions and configurations is essential to each of these domains, requiring the environment to capture and organize both large-scale and small-scale updates. *Coarse-grained changes* provide the basis for release and configuration management and are conventionally handled by a source code control system. *Fine-grained* changes are typically managed by the 'undo' and recovery activities of an editor. The two levels are combined in our approach: the timing and granularity of checkpoints is a policy decision.

We build on theoretical results developed for persistent linked data structures to create *self-versioning documents*. These documents cache their current contents, just as conventional representations do, but also provide access to previous versions by recording modifications as they occur and indexing the history of changes. The approach is designed to support high-bandwidth, fine-grained recording: individual textual and structural modifications as well as complex transformations (e.g., incremental compilation) can be treated as atomic updates.

This chapter covers the implementation of lightweight versioned objects and a document representation that utilizes these objects to provide fine-grained versioning for main memory history logs. The design is object-based: the primary index for updates is *spatial*, with each object encapsulating its own history by recording the association between modification times and values. This is in contrast to the usual design of editor 'undo' logs, where the primary indexing method is temporal (and often limited to a single entry). Our work augments techniques for object caching and off-line storage developed for object-oriented databases [13] and for multi-user locking and nested transaction support [64].

Recording modifications at this level of granularity requires attention to bandwidth constraints and representation issues: a typical document has many nodes, each containing several versioned fields. Existing methods for capturing document updates (such as `rcs` [91]) are too slow and heavyweight to support fine-grained capture, and lack crucial support for structural representations and multimedia elements. Our approach is based on efficient methods for producing persistent linked data structures and is fully incremental: current values are accessible in constant time and non-current values in $O(\lg |\text{local modifications}|)$ time, with minimal overhead in both cases.

The representation described here has been used as the underlying document representation in *Ensemble*. Measurements of this system indicate that less than 4% of its running time is attributable to versioning. Recording a change to most versioned datatypes requires only 34 additional bits of storage. Caching policies and inlined query methods enable the current document content to be accessed as quickly as in unversioned documents. Lazy history log instantiation optimizes storage for unmodified objects.

The interface provided by the versioning scheme is both simple and largely media-independent.[1] Objects transparently

[1] We do not describe multimedia encoding methods; formats such as MPEG are well-known [33], and Section 3.5 describes data structures for versioning such datatypes as pointers, booleans, and large text buffers.

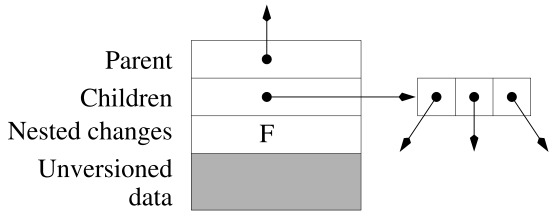


Figure 3.1: Document node representation. Each node typically contains both versioned and unversioned fields. The latter include read-only data and any value whose lifetime is global or restricted to a single update.

record the modifications made to them, stamping each change with the current 'global' version number. Each object is fully persistent (can provide its value for any existing version). Access to the current value is optimized. Self-versioning documents are built simply by using versioned objects for the edges (link fields) in document nodes. Data fields can be versioned as well; the nodes themselves retain their identity to simplify reference maintenance. Any version of the document can serve as the basis for a new version.

In addition to a low-level versioning approach that enables us to record modifications and answer queries about the history of a document component, we provide a flexible *change reporting* framework that allows clients to efficiently discover the regions of a document modified during some time period. The interface supports unrestricted queries, since different clients will have different needs. (For example, the presentation services will need to update the on-screen image of a program after every keystroke, but the user may wish to make several changes before incrementally recompiling the executable image.) Only a single versioned boolean field per internal document node is required to enable change reporting.

The remainder of this chapter is organized as follows. In Section 3.2 we describe our document model and the client interface to versioned objects in greater detail. In Section 3.3 we examine two concrete implementations of self-versioning documents: programs and essays. Section 3.4 describes the run-time representation of the version hierarchy and its role in grouping updates. The implementation of versioned objects is described in Section 3.5.

3.2 Document Representation and Services

Although versioned objects can be instantiated individually, they most commonly occur as fields within document nodes. When a graph is constructed using versioned fields, the data structure as a whole will be versioned. Document nodes can contain other types of versioned data in addition to links. In *Ensemble*, nodes are instances of C++ classes containing both versioned and unversioned fields.

For the applications of interest to us, documents are primarily represented as trees, although dags and more general graph structures can also be modeled. Interior nodes provide structure while terminal nodes typically contain media-specific content (text, graphics, etc.). (Figure 3.1 illustrates an interior document node.) Information associated with a document node can be represented as field values, annotations, or entries in a separate database. The relationship between structure, content, annotations, and database entries may be maintained in whole or in part by automatic mechanisms. In the case of a program, for example, incremental analysis and transformation mechanisms preserve the relationship between the abstract syntactic structure, the textual content, and the binary representation of the compiled program. Although multilingual/multimedia documents are supported both by the model and by *Ensemble*, in this work we will assume a single language for the entire document (the structure can thus be described by a context-free grammar) and primarily text-based content.[2] We assume a single current (cached) version for each document, although the techniques can be generalized to support multiple cached versions.

Our primary applications are highly interactive, requiring incremental algorithms and efficient access paths to all regions of a document. To guarantee logarithmic access to each document node, we augment the document's intrinsic structure with a balanced representation of *sequences* of document components. Lists of paragraphs, graphical objects, statements, and so forth are represented internally as balanced binary trees that are incrementally rebalanced at each *commit point* (see below). A balanced sequence representation allows both natural and formal language documents to be stored, accessed, and transformed using efficient incremental services.

In a typical scenario, a document is modified repeatedly using one or more tools. Modifications are grouped by the client into atomic updates using begin/end edit methods of the global version tree (Section 3.4). Our approach makes no assumption regarding the frequency of updates; in fact, the finest level of granularity consistent with user expectations and interactive performance should be provided. When combined with high-performance object-oriented database technology,

[2]Extensions to the *Ensemble* document model to support compound documents and a discussion of issues involving multimedia elements and natural language documents are described by Maverick [65].

```
<T> get ()
void set (<T> value)
bool changed ()
bool changed (from_version, to_version)
void alter_version (version_id)
bool had_value (<T> value, from_version, to_version)
bool exists ([version_id])
void discard ()
void mark_deleted ()
void undelete ()
```

Table 3.1: Interface to low-level versioned objects. Several of these functions are expressed as templates; an instantiation is provided for each versioned datatype. Arguments in square brackets are optional. Versions are identified by clients using an opaque type, which is implemented as an integer. (Its value is simply the index of the version in the creation sequence.)

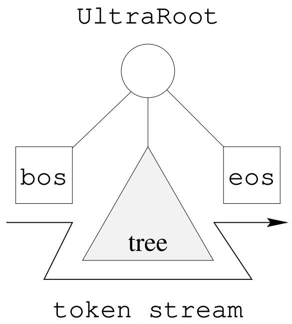


Figure 3.2: The relationship between the three permanent sentinel nodes and the document's structure. Two permanent tokens bracket the terminal yield of the tree, while a third sentinel (the `UltraRoot`) points to both of these tokens as well as the current root of the document tree.

the techniques described here enable a seamless integration of source code control, editor undo logging, and filesystem caching for software documents [34], and a similar degree of support for natural language documents (for which commercial application programs rarely provide useful history-related services).

The document's own structure (along with any imposed tree structure for sequences) is used as an 'implicit' spatial indexing scheme for many version-related operations. These operations include answering client change queries (see below) and altering the version of the entire document, which uses the same change reporting interface internally. Versioned data fields in document nodes are automatically synchronized with the version of the document as a whole.

Analysis and transformation tools discover document changes by performing a tree traversal that is restricted to only those areas that have been modified. Queries from different tools need not be synchronized with one another. An individual node is generally considered changed whenever *any* of its versioned fields have been modified since the tool's previous interrogation. To enable efficient traversal of only the updated regions of a document, tools must also know whether there are additional changes within the subtree rooted at a given interior node. This *nested change* information is summarized by a versioned boolean field in each interior document node that indicates whether the subtree the node roots (excluding itself) was changed. These bits provide a 'trail' to all the local changes for every version of the document.[3] Nested change bits are ignored during local change queries and are cleared prior to each document update. Unchanged values are not explicitly recorded for any datatype; storage for change bits is thus naturally minimized by editing locality. Multiple edit categories can be supported by using several nested change bits per node.

Table 3.1 contains the basic interface to a versioned object; this API will be described in greater detail in subsequent sections. The portion of the document node interface relevant to history-based queries and versioning is shown in Table 3.2. (Functions to handle transient and undefined values and other miscellaneous state management operations are omitted.)

Tools in the ISDE use permanent *sentinel* nodes to locate starting points in the mutable tree structure. Three sentinel nodes, shown in Figure 3.2, are used to mark the beginning and end of the token stream (`bos`, `eos`) and the root of the tree (`UltraRoot`).

To create a program from a textual representation, a null tree corresponding to only the sentinels in Figure 3.2 and an empty ('completing') production for the start symbol of the grammar is constructed. The initial program text is assigned

[3] Nested change bits represent document *transitions* rather than document values *per se*, and thus require specialized query methods. Appendix A contains the algorithm to compute the status of a boolean object representing nested change information between two points in time.

`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 subtree, 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 link, 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 field.

`void discard(and_nested?)`

Discards any uncommitted modifications to either this node alone or in the entire subtree rooted by it when `and_nested?` is true.

`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.

`void mark_deleted()`
`void undelete()`

Used to indicate that a node has been removed from the tree (or to reverse the decision). `undelete` can only be used prior to committing the deletion.

Table 3.2: Summary of node-level interface used by incremental analyses. Each node maintains its own version history, and is capable of reporting both local changes 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 object.

Find root of each deleted substructure.

```
void process_deletions (NODE *node) {
  if (!node→is_new && node→has_changes(local))
    for (int i = 0; i < node→arity; i++) {
      NODE *old_kid = node→child(i, previous_version);
      if (old_kid→exists() && !node→has_child(old_kid) &&
          (old_kid→parent() == NULL || !in_tree(old_kid)))
        old_kid→handle_deletion();
    }
  if (node→has_changes(nested))
    for (int i = 0; i < node→arity; ++i) process_deletions(node→child(i));
}
```

Mark contiguous deleted nodes.

```
void handle_deletion (NODE *node) {
  int i;
  Copy children so we can mark the versioned objects in this node deleted.
  NODE *kids[node→arity];
  for (i = 0; i < node→arity; ++i) kids[i] = node→child(i);
  node→mark_deleted(); Future calls to node→exists() will return false.
  Iterate over kids, checking each one for deletion.
  for (i = 0; i < node→arity; ++i)
    if (kids[i] && kids[i]→exists())
      if (kids[i]→parent() == NULL || kids[i]→parent() == node)
        handle_deletion(kids[i]);
      else if (!node_in_current_tree(kids[i])) handle_deletion(kids[i]);
}
```

A node is in the current version of the tree if a retraceable path to the root exists.

```
bool node_in_current_tree (NODE *node) {
  if (!node→exists()) return false;
  for (NODE *p = node→parent();
       p != UltraRoot && p→exists() && parent→has_child(node);
       p = p→parent()) node = p;
  return node == UltraRoot;
}
```

Figure 3.3: Processing deleted nodes. This algorithm locates unreachable nodes—nodes present in the previous version of the tree but not in the current version—and marks them as deleted. The search is performed in $O(d + s \lg N)$ steps for d deleted nodes, s modification sites since the previous commit, and N total nodes in the new tree.

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.

At commit time, each versioned object that is no longer in use must be placed in a *deleted* state to indicate the end of its lifetime. Deleted objects are contained in the set of deleted nodes—nodes present in the previous version but not the current version. The set of deleted nodes is discovered at commit time by the algorithm in Figure 3.3. It uses the standard client change reporting interface internally: each structural modification indicates a location from which nodes may have been removed. Deleted structure is recursively investigated (by traversing the previous structure of the document) until it merges with retained structure or a leaf node is reached. The `mark_deleted()` method is invoked on each deleted node. (Nodes are not deleted in the C++ sense until the document as a whole is discarded or the user indicates that the range of versions that constitutes a node's lifetime is no longer needed.)

3.3 Applications

We consider two examples from *Ensemble* to illustrate self-versioning documents: programs and simple natural language documents ('essays') composed of text and graphics.

3.3.1 Programs

Each program module is represented as a document; the document's structure corresponds to the program's abstract syntax tree. Tokens are represented by terminal nodes that contain text. (Stream-style comments and explicit whitespace are integrated with the structure of the program; see Appendix C.) Associative sequences, such as declaration and statement lists, are identified in the grammar using regular expression sequence operators (Section 6.6).

In *Ensemble*, each character inserted or deleted creates a new version, as does any structural modification applied directly by the user. More complex structural reorganizations, carried out with the aid of tools, apply multiple changes within a single version. When the user requests, consistency is restored to the program representation using a collection of incremental analysis/transformation algorithms, which are supplied with the set of accrued changes since their previous invocation. (The model for editing, analyzing, and transforming programs using language-based tools is discussed in Chapter 4.)

Programs represent a stress case for fine-grained versioning because the structure itself is both large and dense (possesses a high ratio of nodes to textual content). Analysis and transformation tools that operate on the program (including such 'tools' as editing and presentation services) require precise change reporting in order to avoid time-consuming recomputation in their own analysis, and to avoid triggering unnecessary work during subsequent analyses. Incremental syntactic and semantic analyzers use the query methods provided by the document node interface (Table 3.2) to discover regions of the program structure that have changed since the previous analysis. Internal details of document nodes and versioned objects are fully encapsulated from the implementation of these and other tools in the environment.

3.3.2 Essays

Natural language documents are 'flat', possessing minimal intrinsic hierarchical structure. As with formal language documents, a context-free grammar describes the permissible structure of this document type to the environment. Balanced binary trees are used to represent sequences of paragraphs. Each keystroke or collection of related keystrokes is normally treated as an atomic update, as are certain 'compound' operations, such as splitting a paragraph.

Essays differ from programs by containing multiple media and text strings of non-trivial length.[4] The string storage method must therefore record substring modifications to avoid wasting space and to support fine-grained change reporting. Such *differential* storage differs from the state-based 'snapshot' recording used for small datatypes, such as links and integers. Note that object-based graphics can be handled without additional mechanisms, since the attributes of these objects (location, radius, angle, etc.) can be represented as versioned real numbers, and the set of graphic objects in a figure can be treated as a sequence.

3.4 The Global Version Tree

Each atomic update creates a new version, derived from its parent version. The *global version tree* (GVT) captures this hierarchical relationship among the versions of a document.[5] The GVT is materialized in our approach: it provides the version naming service to document clients and serves as a shared portion of the implementation of the versioned objects.

The GVT also serves as the focus of transactional control. Clients use explicit begin_edit/end_edit methods on the GVT object to define each version and associate it with a set of modifications. Each call to `end_edit` indicates a commit point.[6] Between the committing of one version and the construction of a new version, clients can use additional GVT methods to change the cached version. (To simplify the exposition here, we assume a single GVT, a single document, and at most one version in progress at any time.) The current version becomes the parent of the new version created when `begin_edit` is invoked.

Figure 3.4 shows a sample GVT containing five versions. The leftmost tree represents the conceptual GVT; on the right, its physical realization as a collection of data structures is shown. Each version is named by a unique integer, which indexes an array mapping ordinal version names to GVT nodes. Within the GVT, *chains*—linear 'runs' of successive versions, each derived from the previous version—are collapsed. The efficient representation of chains minimizes the size of the GVT and speeds the calculation of values (other than the current value) within versioned objects. A new GVT node is added as the *leftmost* child of its parent. Both pre- and post-order sorted lists of the GVT nodes are maintained to support efficient 'ancestor-of' queries.[7] Each versioned object's private history is a subset of the global version tree; an efficient array-based representation for these local version 'trees' is described in the following section.

[4] In a program, tokens average only a few characters in length [102]; in an essay, however, each 'token' typically represents an entire paragraph.

[5] We do not consider the merging of different versions here. In coarse-grained merging individual versioned objects rarely conflict; in cases where they do, the resolution is necessarily media-specific.

[6] Commit-time operations, such as rebalancing sequences and computing the set of deleted nodes, are performed at this time.

[7] The pre-order sort of the GVT corresponds to the *version list* in Driscoll et al. [26]. The implementation of these lists is not shown in Figure 3.4.

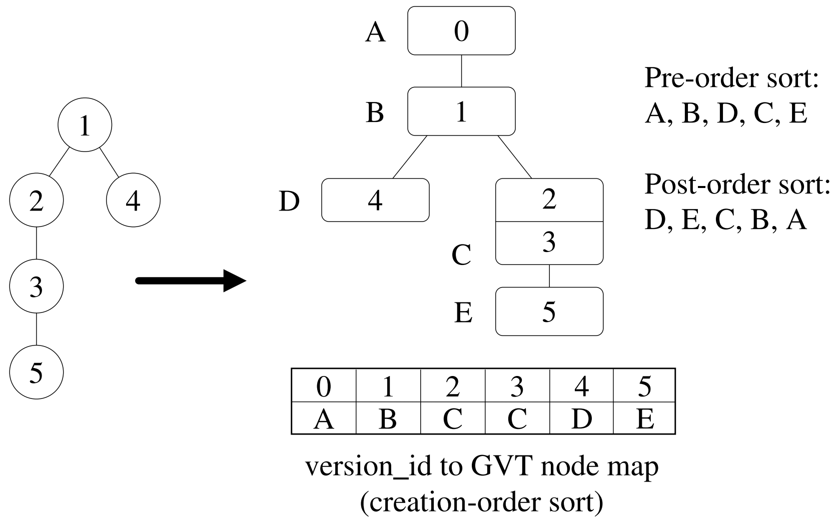


Figure 3.4: Global version tree. The version hierarchy is defined by the ancestor relationships between the nodes. The figure on the left illustrates the conceptual relationships among the versions. The figure on the right shows the actual implementation, which differs in several ways: the addition of a sentinel version, ordering of the children, compression of linear runs, and the three sorted lists (pre-order, post-order, and creation-order) to enable efficient name lookup and ancestor-of tests.

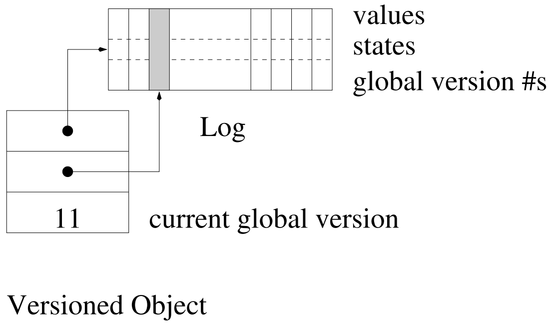


Figure 3.5: Representation of a versioned object. The `current_global_version` field can typically be eliminated as an optimization.

As with document structure, a sentinel root node (version 0) is used to simplify the implementation of algorithms that manipulate the GVT. Version 0 also has a convenient semantic interpretation as 'pre-historical' time, which is useful in bootstrapping documents created from outside sources.

3.5 Implementing Versioned Objects

In this section we describe the data structures and algorithms required to implement the versioned object interface described in Section 3.2, while meeting the time and space requirements of interactive applications. The theoretical basis for the design is a corrected version of the 'fat node' approach to persistent linked data structures developed by Driscoll et al. [26].

3.5.1 Full-State Storage

Many of the elements of a self-versioning document, such as the links between the nodes themselves, are small datatypes that are treated as intrinsic values. These datatypes are versioned by recording their values (when changed) at each checkpoint; the historical representation is conceptually an array of <*version*, *value*> pairs. The versioned object itself is implemented as a pointer to this log, coupled with one or more log indices that denote the current (cached) values of the object. In addition to the global version identifiers and the values themselves, the log also contains state information, allowing deleted and undefined object states to be supported. Figure 3.5 illustrates the implementation of a typical versioned object.

A versioned object's log can be implemented in a variety of ways. *Ensemble* uses dynamically sized arrays to minimize

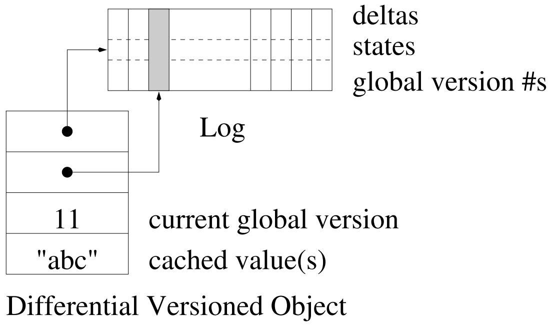


Figure 3.6: Representation of a differential versioned object. In this case, the log entries store *deltas*, instructions to create the child version from its parent and vice-versa, instead of values. The current value is cached with the object itself.

space overhead.[8] For additional time efficiency, the log can be maintained as a pair of arrays stored in contiguous memory with an 'edit gap' between them. This arrangement is consistent with typical application requirements: documents are frequently changed at multiple points, but it is less common for the user to back up and begin modifications from a previous version. To alter the current version of a full-state object, the index field of the object is simply changed to point to a different value in the log. Section 3.5.3 describes the mapping of global versions to local log indices.

Recall from Section 3.4 that the structure of the global version tree is partially defined by the order of version construction, which establishes the parent⇔child relationships among the GVT nodes. The Driscoll algorithm produces a complete ordering by arranging sibling GVT nodes left-to-right by *decreasing* version numbers. The log entries are sorted by their global version fields, ordered according to the pre-order linearization of the nodes in the global version tree. When a new value is recorded in the object, an *additional* entry is needed if the version following the current global version in the the pre-order sort is not already represented in the local log.[9]

3.5.2 Differential Storage

Some datatypes, such as lengthy text strings, audio and video recordings, etc., are too large to store in full at each checkpoint. Instead, only the *difference* from one version to the next is stored. Although representation details vary from one medium to another, the framework for accessing and manipulating 'differential' objects is identical.

As with other versioned objects, the fundamental storage model is a log, in this case recording *deltas*, rather than complete values, for each changed version. An entry in this log describes how to build the local value associated with a particular version by applying the stored delta to the object's value in the parent version.[10] Deltas must also be reversible, in order to build the parent version's value when the object is in the child version's state. The current value is cached in the versioned object itself (Figure 3.6).

To query the object's value at a different point in time without changing the cached value, a temporary copy is made and then progressively modified into the target version by applying deltas from the (shared) log.

3.5.3 Projection

The most frequently requested value for a versioned object is its current (cached) value. To produce the value corresponding to a different time requires mapping the name for the target global version (its position in creation order) into a name for the local version—an index into the object's local history log. (This mapping thus *projects* the global version onto the local history.) The projected index is the rightmost one such that the global version recorded for that entry is not to the right of the target version in the pre-order linearization of the GVT. The `project` algorithm in Figure 3.7 computes this result efficiently using a logarithmic search of the object's log, keyed to the global version field.

The compact form of GVT nodes, which collapse linear runs in the GVT, optimizes the comparisons performed by the projection algorithm in the common case. When two global versions belong to the same GVT node, the pre-order comparison is implemented as an integer comparison on the values of the two keys. When global versions belong to *different* GVT nodes, the pre-order comparison can be made efficient by representing the GVT's pre-order sorted list as a data structure supporting $O(1)$ order queries [25].

[8] An implementation based on balanced binary trees can be used to guarantee logarithmic access time to each local value at a modest increase in space overhead.

[9] Driscoll's original test for determining the need for a second entry is flawed: $i+ < i_2$ should be replaced with $i+ \not< i_2$. This change prevents incorrect results in projecting global versions onto the local log; with the change, the correctness proof provided in the original paper is valid.

[10] Any additional log entries required by the Driscoll algorithm are represented as empty deltas.

Map a global version to a local version (log index).
```
int VObject::project (int version) const {
  int l = 0, r = RightmostLogIndex;
  int x, result;
  const GvtNode *gd1, *gd2;
  gd1 = gvt→VersionToGvtNode(version);
  do {
    x = (l + r) / 2;
    gd2 = gvt→VersionToGvtNode(versions[x]);
    Use integer comparisons within a GvtNode
    if (gd1 == gd2) result = version - versions[x];
    else result = gvt→PreorderCompare(gd1, gd2);
    if (result < 0) r = x - 1;
    else if (result > 0) l = x + 1;
    else return x;
  } while (l <= r);
  return r;
}
```

Figure 3.7: Projection algorithm.

Datatypes that employ full-state storage use the projected index to return the appropriate entry of their value array as the result. Objects stored differentially require additional computation, since multiple deltas must be applied to the currently cached value to produce the target value. Repeated calls to `project` are made as the value computation traverses the path through the local version tree corresponding to the path in the GVT between the current and target versions. (See the function `alter_version` in Appendix A.)

3.5.4 Lazy Log Construction

In a typical application, the contents of the document will be loaded into main memory from a disk or network image and subsequently modified by the user. For all but the smallest documents, however, many versioned objects will remain unmodified—their initial value will be maintained until the in-core representation is destroyed. This observation suggests that a log representation, while appropriate for the general case, adds unnecessary space and time overhead to the majority of versioned values.

Instead, we optimize space consumption by initially instantiating all versioned objects as single-valued *temporary* containers. These are conceptually single-entry logs, but lack the arrays and corresponding overhead. If a temporary object is subsequently modified, it is first transformed into a normal log containing the (sole) value; the temporary container is then destroyed, and the modification continues using the newly created log. The use of a temporary representation for a versioned object can be detected by setting its index field to a negative value.

3.6 Conclusion

We have described self-versioning documents that support fine-grained recording of their modification history using an object-based mechanism. The approach is useful for representing programs in software development environments, on-line hypermedia documents, and time-varying data within a database. The techniques are based on an efficient theoretical model, augmented with special support to make the common cases fast and to minimize overall space consumption. Highly efficient change reporting is provided by adding a nested change summary bit to internal document nodes. The combination of intrinsic document structure and a balanced binary tree representation of sequences enables clients of this representation to achieve incremental performance when navigating or modifying the document.

Chapter 4

Integrating Incremental Analysis with Version Management

In the previous chapter we introduced self-versioning documents as the basis for a persistent, structured representation of programs, where the history of changes is spatially distributed throughout the tree. Later chapters describe specific algorithms for maintaining the consistency of the program structure through incremental lexing and parsing.

In this chapter we bridge these concerns. We first introduce the language object model, which extends the basic environment services with information and algorithms specific to a particular language. We then describe a general model for editing programs, using the change reporting mechanism supplied by self-versioning documents to inform incremental analysis tools of the modified areas, which in turn determine the portions of the program that require inspection. Language-based transformations are themselves captured in the history log, providing uniform reporting and full reversibility of even the most complex updates.

Changes made by a tool are reported to any subsequent analyses; when such changes involve the program structure, it is essential that each tool minimize its impact by avoiding unnecessary changes. A strategy of information preservation through node reuse decreases total response time, saves space in the history logs, and improves the user interface by eliminating spurious updates.

4.1 Introduction

In formal language documents such as programs, important subsets of the relationships among content, structure, attributes, annotations, and database entries are derived and checked automatically. This *consistency maintenance* is accomplished through analysis and transformation algorithms, which modify the document by replacing existing components or constructing new components in order to integrate the user's changes.

Batch compilers read characters from a persistent storage device and construct various temporary data structures to perform their work. In an incremental SDE, the program representation serves a dual function: it records the user's changes *and* it serves as the target of the language-based transformation. It is the need for this single data structure to serve as both the input and output of the transformation that makes the self-versioning document model so powerful: the model provides efficient, fine-grained access to multiple versions *simultaneously*.

Ensemble's analysis and transformation algorithms operate *incrementally* to provide scalable performance and interactive response times. This fundamental requirement extends to the services used by these tools, specifically the discovery of changes to the document by the user and the recording of changes made in the course of applying a language-based transformation. Due to their limited document models and batch tools, conventional software development environments need not integrate analysis algorithms and history services. In an ISDE, however, the lack of a history mechanism would require *every* analysis tool to track user edits (as well as relevant changes by other tools) and to summarize the set of pending changes on demand. Our approach generalizes and integrates such 'change tracking' services, simplifying the implementation of all tools and improving overall performance. In addition, the uniformity of change recording and reporting allows *any* transformation to be undone using conventional editor services. Generic development services, such as computing differences between versions or checkpointing the current environment state, treat updates identically, regardless of their source.

The self-versioning document framework makes analysis/transformation algorithms easier to implement and to combine. The history interface provided by this framework allows each version of the document to appear distinct, making it possible for a single tool to read from several versions of the document in order to create a new version *without regard to*

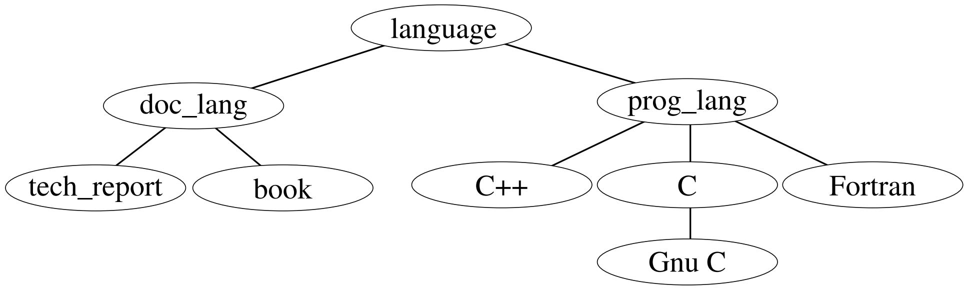


Figure 4.1: Class hierarchy for run-time language objects.

the relationship among the versions.[1] Using the history mechanism as an inter-tool protocol to report changes *encapsulates* transformations: the relative order in which tools read and write the document structure is immaterial. The versioning system also enables a clear separation of functionality: different transformations can be performed at different times and each can discover the exact set of changed components since its previous execution. There is no *a priori* commitment to the number, frequency of application, or relationships among the environment services.

The shared nature of the document representation in an ISDE imposes certain design considerations on the tools that modify this data structure. Information associated with the document—profiles, hyperlinks, comments, semantic attributes, and other information crucial to software development—should be preserved across any transformation that does not logically remove the component to which the information is attached. Previous incremental algorithms have largely ignored this fact, and often generate new structure instead of reusing existing elements. Regeneration fails to preserve user context (such as debugger breakpoints, comments, and profile information) as well as tool context (cached tool data used for incremental analysis). Although retained information may have additional dependencies that render it invalid in the transformed document, maintaining the association when possible provides greater flexibility and typically results in better end-to-end performance and improved user services.

This chapter is organized as follows. We first complete the discussion of document representation by introducing the *language object model*: the run-time representation of language services derived from formal specifications. Section 4.3 describes a generalized model of user editing and its integration with incremental analysis algorithms via the change reporting service. Section 4.4 discusses the *analysis model*, in which incremental tools use the history mechanisms to gain access to multiple versions of the program in order to restore consistency among the document's components. Section 4.5 discusses the impact of transformations on subsequent analyses and the user, and how information can be preserved through node reuse.

4.2 Language Object Model

Languages are specified via formal definitions, which are compiled off-line and loaded into the running environment. Languages are organized into a (relatively flat) hierarchy that permits the relationship between dialects, vendor-specific additions, and different language versions to be expressed directly. An explicit hierarchy also provides a clean method for establishing default behavior for both natural and and programming language documents that can be overridden as needed. Figure 4.1 illustrates a sample hierarchy; Table 4.1 contains the interface to the base language class.

As in batch environments, certain analyses and transformations are needed by every formal language: lexing, parsing, semantic analysis, etc. These services are complex, more so than in batch systems due to the added complexity of incrementality. To simplify the description of new languages to the environment and to reuse existing implementations and correctness proofs, we use *generic* run-time mechanisms. These take the form of methods associated with the class of formal languages, which implement the language-independent logic for each analysis/transformation tool. Language-specific customization is produced by compiling a high-level specification meta-language, such as a grammar.[2]

The language hierarchy enables the replacement of generic analysis services on a per-language basis when some feature of a particular language renders the generic algorithm unsuitable. Inheritance allows a natural formulation of multiple evaluation strategies; for example, Chapters 6 and 7 describe two incremental parsing strategies optimized for deterministic and

[1] In the absence of an explicit history mechanism, each language-based tool must simulate multiple-version access and non-destructive updates of unread document structure, complicating its implementation and potentially exposing its internal invariants to other tools.

[2] Novel descriptive formalisms for presentation and semantic analysis have been developed as part of *Ensemble*. Meta-language notation for the lexical and context-free syntax was not a specific focus of our research, which reuses established formalisms [18, 75]. An integrated language specification language that permits the automated derivation of efficient incremental evaluators remains an area of active research [96].

<LanguageClass> (filename)
: Construct a new instance of a language class by dynamically loading a compiled language specification in the form of a shared library.

LanguageObject *copy ()
: Construct a new instance of a language class by duplicating an existing instance.

String name ()
: Retrieve the name of the language/dialect.

String release ()
: Retrieve the release (version) number of the language/dialect.

Grammar grammar ()
: Retrieve a (shared) copy of the grammar.

void update (ultra)
: Restore consistency to the program representation associated with this language object instance by invoking one or more incremental evaluators.

Table 4.1: Run-time interface to language objects. Additional functionality may be provided for the class of natural or formal languages. Each language customizes the implementation of the `update` method by specializing implementations of the analysis/transformation tools, which may be further refined for individual dialects or versions of the language.

non-deterministic syntax, respectively. The language object model also provides a clean integration with *ad hoc* procedural code by permitting it to replace a generated tool.

Once a compiled description has been loaded into the environment, separate instances of the language class are constructed for each document expressed in that language. (The language instance is a field of the `UltraRoot`; see Figure 3.2.) Every language instance includes instances of each of the language-specific tools, which are applied to the document to provide the language-based services. Much of the code and data is shared among these instances; for example, all parsers for the same language share a single copy of the (read-only) parse table. Data structures required for evaluation, such as the parse stack, are created on a per-instance (and therefore per-document) basis.

Each language instance also shares a copy of the grammar. Since tree nodes are instances of productions, each production in the grammar is translated into a C++ class inheriting from a common `NODE` type. (Terminals inherit from a more specific class, `TOKEN`.) These classes are automatically generated when the language description is compiled. Other tools, such as semantic analysis, can further specialize the representation of productions by adding additional fields and/or methods [41, 63]. The grammar provides a constructor routine for each production (node class). Information can also be associated with nodes via annotations or as references from a program database, both of which operate on the generic `NODE` type and therefore require no special customization for a particular language.

4.3 Editing Model

We permit an unrestricted editing model: the user can edit any component, in any presentation, at any time. The user's 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.[3] Between incremental analyses, the user can perform an unlimited number of mixed textual and structural[4] edits, in any order, at any point in the program. Incremental performance is not adversely affected by the location of the edit site(s)—changes to the beginning, middle, or end of the program are integrated equally quickly. The user can also create or modify explicit annotations on document nodes and, depending on the policy and environment tools, may be permitted to directly update a subset of the semantic attributes or database entries as well.

[3] This policy reflects experience showing that re-analysis after every keystroke is unnecessary for adequate performance and the (typically invalid) results will be distracting if presented to the user [93].

[4] 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 (see Section 8.5.1). Each textual modification is represented as a local change to the terminal symbol containing the affected characters.

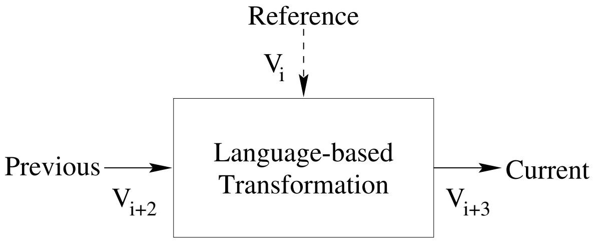


Figure 4.2: Language-based document transformations. This figure illustrates the general form of an incremental analysis/transformation of a formal language document. The *previous version* of the document (left) is transformed into the *current version* (right) by the application of an incremental tool. Language-based tools, such as incremental parsing, also require a *reference version* (top) that represents a state of the document where the structure is consistent. The difference between the reference and previous versions determines the document components that are potentially reusable. As a special exception, the initial analysis of a newly entered program has no reference version, since it represents a batch scenario. (The version numbers shown correspond to the example in Figure 4.3.)

The self-versioning document model handles all transformations, including both user changes and those applied by tools such as incremental parsing, in a uniform fashion. Changes made when a tool performs a language-based transformation are captured using the same history mechanism that records user updates, and propagated to other services through change reporting. In addition to providing a rational user interface, the integrated treatment of updates provides a novel capability in the form of full *reversibility*: the same interface used to undo textual and structural edits can be used to undo the effects of incremental lexing, parsing, error recovery, and other complex transformations.

4.4 Analysis Model

An ISDE includes a variety of tools for analyzing and transforming programs. The persistent program structure, in the form of a self-versioning document, is the central data structure shared by these tools.

'User edits' are distinguished within the environment as document transformation that are not guaranteed to maintain the invariants among a document's components, resulting in potential inconsistencies. When the user chooses to restore consistency, language-based analysis and transformation tools are applied to restore the maintained relationships by incrementally modifying the document to produce a new, consistent document state.

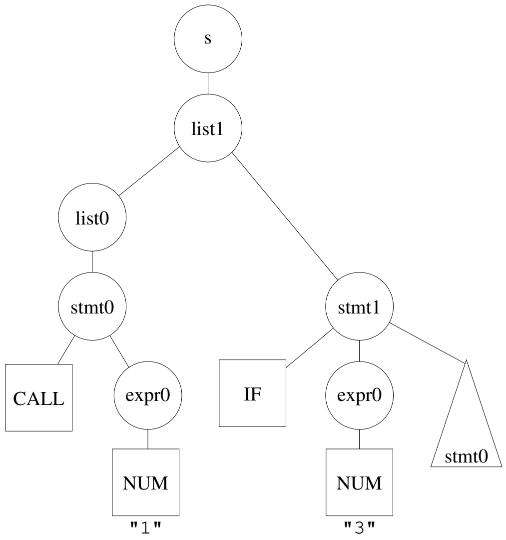
The history mechanism described in Chapter 3 provides the protocol by which different transformation and analysis tools communicate their effects. The interaction between analysis-based transformations and history services follows a general form, illustrated in Figure 4.2. The tool examines (a portion of) the document representation and applies zero or more changes to restore consistency for the relationships it manages. Destructive modifications of the document structure, content, annotations, etc., are captured by the history mechanism in the same fashion as user edits. Language-based transformations typically compare the previous state to another consistent state; the difference determines the set of reusable components. The three document versions are referred to symbolically:

Reference: A version of the program that represents a consistent state (one constructed through an analysis/transformation that restored consistency).[5]

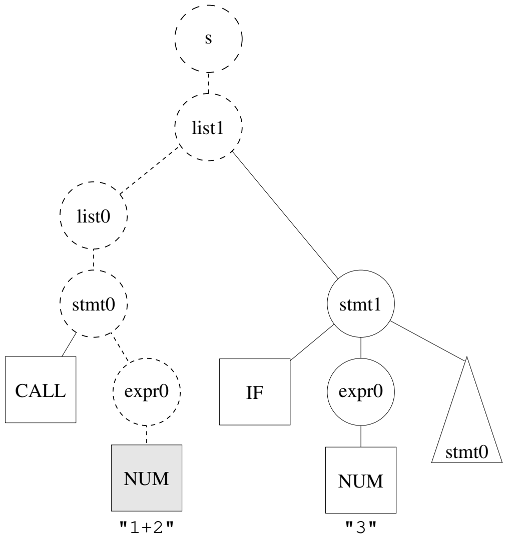
Previous: The state of the program immediately prior to the start of re-analysis. This is the version read by the incremental tool in order to perform its analysis. (The modifications accrued between the reference version and the previous version determine the potentially reusable material, and thus constitute the inherent limit on incrementality.)

Current: The version being written (constructed) by the incremental tool. This constitutes another consistent version that can potentially serve as the reference version for a future application of this tool.

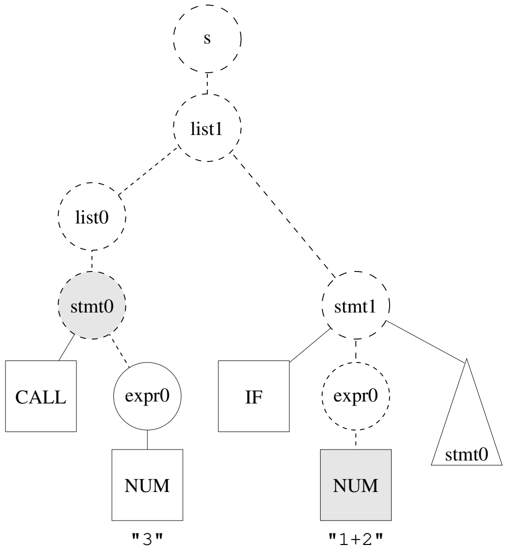
The analysis model allows significant flexibility: multiple tools may act cooperatively to produce a new version, or they may be applied sequentially, producing intermediate (committed) versions that are logically grouped into a compound transformation by the user-level version management interface. Figure 4.3 illustrates a simple example. Two user modifications (one textual, one structural) are processed by incremental lexical and syntactic analysis. The result can either be committed or the analysis can continue by applying semantic analysis or other tools before reaching a final result. The resulting sequence of versions is shown in Figure 4.4. Recall from Chapter 3 that the conclusion of each atomic update sequence is



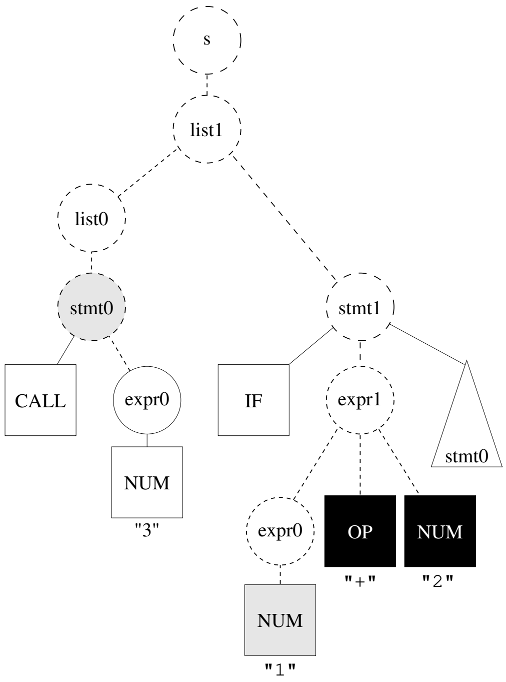
3a. Initial state of the program: the text and structure are consistent.



3b. After a text edit to the first number (from “1” to “1+2”).



3c. After a structural edit (swapping the two expression subtrees).



3d. After incremental lexing and parsing. Two new nodes have been created; all others continue to function in their previous roles.

Figure 4.3: Sample editing sequence, starting from a consistent state, (a). In (d) an intermediate point in the analysis is shown: incremental lexical and syntactic analysis have completed their transformations to the document structure, and semantic analysis is now preparing to restore consistency to the semantic attributes. Gray indicates local changes, black indicates new nodes, and dashed lines indicate paths to modification sites. Figure 4.4 illustrates the sequence of versions in this example.

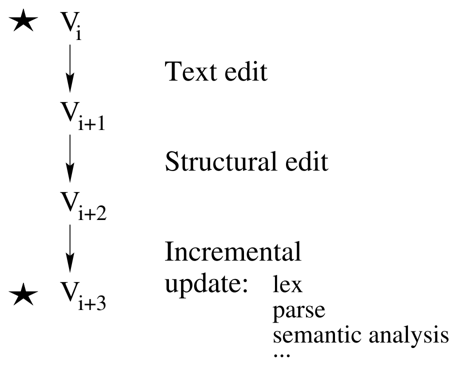


Figure 4.4: Version sequence for Figure 4.3. The starred versions represent consistent states.

indicated by a call to `end_edit` that commits the changes; once committed, the version becomes read-only.[6]

Apart from incremental lexing and parsing, which maintain the persistent document structure, the choice to version or recompute tool-specific data is a policy decision. Placing data under control of the history services causes it to be updated 'automatically' if the user changes the current version of the document, but requires slightly more space to represent than unversioned data. In *Ensemble*, semantic attributes are cached in *unversioned* fields; these values are recomputed (on demand) if the current version of the document is changed or when the document structure has been updated since semantic attributes were last computed.[7] Presentation information is also computed on demand, with the exception of explicit user overrides and attributes set by error recovery (Chapter 8), which are logically part of the document and are therefore represented as versioned annotations.

4.5 Node Reuse

Within an ISDE, many analysis and transformation tools cooperate to provide incremental compilation and associated environment services. The overall performance of the environment is therefore affected not just by the speed of a tool's individual performance, but also by the impact of its changes on *other* tools.[8]

Minimizing unnecessary updates is especially important for tools early in the 'pipeline', specifically incremental lexing and parsing. These tools have the primary responsibility for maintaining the central data structure in the environment—the program's structural representation. Other tools typically establish one or more maps from nodes in this structure to their internal data, associations which are used during analysis to achieve incrementality. The manner in which the incremental lexer and parser integrate changes while transforming the program's structure is thus important. Updates should be precise, since spurious updates unnecessarily inflate the changed set for subsequent analyses. *Reusing* a physical tree node instead of destroying and re-creating it can result in significant savings in time and effort for tools further down the analysis pipeline.[9]

In the case of semantic analysis in particular, a subset of the node attributes will represent the distributed symbol table; loss of this information can require significant recomputation time, due to its non-local nature. While node reuse by the lexer or parser cannot *guarantee* that semantic or other attributes of a given node remain valid, the destruction of the node can only increase the cost of maintaining any associated information. Reuse is especially important for nodes high in the tree (the so-called 'spine' nodes on the path from the root of the tree to the modification sites), since such nodes represent both the major syntactic units of interest to the user and the likely targets for associated tool data.

Node reuse also has a strong impact on the user interface. In a visual presentation of the changed portions of the program, inexact or spurious modifications by the environment tools are confusing, since they violate the user's intuition. Unnecessary node reconstruction may also discard annotations created by the user, forcing him to manually restore any such associations. (Such annotations may also be generated by a time-consuming process, such as profiling; failing to reuse nodes in this setting can result in lost information that is difficult to replace.)

Chapters 5 through 8 discuss specific implementations of reuse techniques in the context of each incremental algorithm we present.

[6] The user's view of versions may be distinct from the sequence of low-level commit operations. Multiple versions may be grouped into a single 'logical' version, which appears to the user as an atomic update. In particular, separating the transformations of multiple tools applied in succession allows their effects to be distinguished: each change will be 'tagged' with the version in which it occurred, uniquely identifying the authoring tool.

[7] Thus the reference version for semantics is not necessarily the same as the reference version for lexing and parsing.

[8] With the exception of Larchevêque's parsing algorithm [58], previously published incremental algorithms have largely ignored tool interaction.

[9] Additional node reuse also saves space by decreasing the size of the history log, since fewer modifications will be outstanding when the version is committed.

4.6 Related Work

Many proposals for handling versioning have been discussed in the database community [52]; the focus has been primarily on coarse-grained updates and non-incremental tools.

Magnusson, et al. describe the sharing and collaborative framework of the Mjølner project [54, 64]. Our concept of integrating analysis-based transformations with history services is compatible with such an approach. Our analyses are designed to maximize component (and information) preservation within structured documents, rather than duplicating paths as the Mjølner system does.

Document processing systems have explored the role of structure in the context of natural language rather than programming language documents [78]. Our approach provides interoperability with these designs, and also suggests a framework in which automated transformations (e.g., page breaking) can be considered as first-class edit operations similar to the incremental analysis performed in formal languages.

Fraser and Myers [34] present a single-level storage model for integrating an editor's undo/redo facility with a source code control system. They use a text-based model (developed from `vi`) where revisions are described in line-oriented terms. Hierarchical document models are a more natural representation for structural changes and support the specialization of media-specific updates.

Larchevêque [58] discusses the motivation for component reuse in the context of an ISDE, and describes how the incremental parser for O_2 was designed to accommodate this goal. Our parsing algorithm provides both a higher rate of reuse and better asymptotic performance (Section 6.7).

4.7 Conclusion

This chapter presented a model for editing and transforming structured formal language documents using the self-versioning representation of Chapter 3. Designing incremental algorithms to operate in this framework exposes their fundamental reliance on multiple versions and the relationship between reuse detection and the preservation of document information. The analysis model permits complex transformations involving simultaneous access to multiple versions of the document. The integration of fine-grained versioning with incremental algorithms is a novel feature of *Ensemble*, simplifying the algorithms by extracting common code for detecting and reporting changes, providing full reversibility of *any* update, and enabling entirely new services, such as history-sensitive error recovery (Chapter 8).

Part II

Incremental Analysis and Transformation

Part I established the fundamental representation of documents in *Ensemble*, and the framework for a language-based analysis and transformation of programs and other formal language documents. In Part II we focus on the specific algorithms that operate within that framework. These tools maintain the persistent program structure, establishing consistency between the structure and content after changes made by the user.

Chapter 5 describes a general model of incremental lexical analysis that improves on existing approaches by generalizing the language model, while maintaining optimal time and space results.

Chapter 6 investigates the incremental parsing of deterministic syntax. Previous results on sentential-form parsing are corrected and extended to provide an optimal time and space evaluator. The handling of lengthy sequences in grammar specification and parse table construction are discussed here.

Chapter 7 extends the language model to include *non-deterministic* syntax, through a generalization of the abstract syntax tree to an abstract *parse dag*. Our incremental analysis model is thus capable of describing and evaluating syntactically ambiguous languages, such as C, C++, COBOL and FORTRAN.

Chapter 8 introduces history-sensitive error recovery, which combines the analyses in Chapters 5 through 7 with the self-versioning document model to provide a new and powerful method for handling program errors.

Chapter 5

General Incremental Lexical Analysis

In this chapter we present the first fully general approach to the problem of incremental lexical analysis. Our approach utilizes existing generators of (batch) lexical analyzers to derive the information needed by an incremental run-time system. No changes to the generator's algorithms or its run-time mechanism are required. The entire pattern language of the original tool is supported, including such features as multiple user-defined states, backtracking, ambiguity tolerance, and non-regular pattern recognition. No *a priori* bound is placed on the amount of lookahead; dependencies are tracked dynamically as required. This flexibility makes it possible to specify the lexical rules for real programming languages in a natural and expressive manner. The incremental lexers produced by our approach require little additional storage, run in optimal time, accommodate arbitrary (mixed) structural and textual modifications, and can retain conceptually unchanged tokens *within* the updated regions through aggressive reuse. We present a correctness proof and a complete performance analysis and discuss the use of this algorithm as part of a system for fine-grained incremental recompilation.

5.1 Introduction

Batch lexers derive a stream of tokens by processing a stream of characters from left to right. Several tools that facilitate the construction of such lexers have been devised, including the well-known Unix tools `lex` [59] and `flex` [75]. These tools support an extension of regular expression notation as their *pattern set*. Each pattern is associated with a rule; in many problem domains the goal is to partition the character stream into tokens, and each rule typically constructs (part of) a token from the text matched by its associated pattern.

In some situations, such as in an ISDE, a series of character streams[1] are repeatedly analyzed with few differences (relative to the total number of characters) between one application of lexical analysis and the next. Since each token typically has a very limited dependence on its surrounding context, the resulting differences in the token stream are typically limited to the area immediately surrounding each modification site. In this setting it makes sense to retain the token stream as a persistent data structure and use it to decrease the time required for subsequent analyses. We refer to the technique of reusing the previous token stream to decrease the time required to produce the new version as *incremental lexing*; a tool that performs this transformation is an *incremental lexer*. We exploit existing technology by combining a specification-derived *batch* lexical analyzer with a novel run-time system that provides the incremental behavior.

Our approach has several advantages. No change to the generator's implementation is necessary. No assumption regarding the implementation of the batch lexer is made. No new descriptive formalism is introduced (*Ensemble* uses the specification language of `flex`),[2] and very few changes to a given lexical description are required to 'port' it to an incremental setting. The resulting incremental lexers exhibit performance competitive with hand-coded implementations.

Our approach is novel in supporting the full expressive power of the underlying tool's pattern language, including user-defined states, multiple tokens per pattern match, multiple pattern matches per token, ambiguity tolerance, and unbounded lookahead. Clean integration with procedural analyses is also permitted on a per-pattern basis.

Our approach also provides extremely fine granularity. By capturing the state of the batch lexing machine at the conclusion of each token's creation and saving it within the token, we are able to restart lexical analysis at any point within the token stream. Restarting the lexer *within* a token is unnecessary; although performance is linear in the length of an individual lexeme, tokens such as whitespace and textual comments can be defined such that newlines or other logical sep-

[1] In an incremental setting we use the term 'stream' to mean the persistent character or token sequence implied by a left-to-right ordering of the terminal symbols in the program structure.

[2] The `flex` specification language is simple, though somewhat low-level. Higher-level or special-purpose lexical specification formalisms can also be used by transforming them into this formalism.

arators serve as token boundaries (see Figure C.2), exposing sufficient granularity for incremental lexing without requiring additional complexity in the generation of the state machine or the size of the token stream.

In general, the mapping between a token and its lexeme is dependent on the surrounding context; a token is *lexically dependent* on the set of tokens that establish sufficient context for determining this mapping. A common, though restrictive, method for handling lexical dependencies in an incremental environment is to place an *a priori* bound on the maximum amount of (textual) lookahead, either by having the language designer provide a limit or by having the lexer generator compute it through analysis of the lexical description. Either way, the expressive power of the language suffers: any fixed bound on the length of a dependency may be insufficient for a particular language. Languages with unbounded lookahead and natural descriptive techniques that similarly lead to unbounded lookahead are precluded by a fixed bound approach. We remove this restriction by tracking dependencies *dynamically*, resulting in a more general language model and potentially faster running times (since the actual dependencies can be used in place of the worst-case assumption).

Dynamic dependencies can be recorded explicitly in tokens at a modest space cost and no asymptotic performance cost. But even for languages or descriptive techniques requiring potentially unbounded lookahead, the vast majority of cases involve a token relying only on the following character. For this reason, we can typically avoid explicit storage by using appropriate *implicit* values, with the exceptional cases represented using a data structure appropriate for sparse annotations. The full power of the specification language can thus be used with no performance penalty and negligible additional space in practice. (For the sake of clarity, lexical dependencies are shown explicitly in the figures in this chapter.)

Our incremental lexing algorithm is optimal, taking $O(c+s \lg N)$ steps for s modification sites in a tree containing N nodes and *c affected characters*—the lexemes of modified tokens and of tokens lexically dependent on any changed token.[3] The batch lexer is invoked the minimal number of times: once for each token in the updated token stream whose contents, lookahead, or starting state may have been modified. No additional asymptotic overhead accrues due to the incremental run-time service; it operates in time linear in the number of old and new tokens containing affected characters.[4]

In addition to permitting the full expressive power of the underlying batch pattern language and unbounded lookahead, our algorithm is novel in addressing the problems of mixed structural and textual edits with arbitrary timing of analyses. We also discuss issues related to the use of incremental lexing within the context of an ISDE, including token reuse, reversibility of the lexer's transformation, and error recovery.

The remainder of this chapter is organized as follows. Section 5.2 discusses related work. In Section 5.3 we provide the background for incremental lexing: the editing model, interface to the batch lexer, and persistent representation of state and contextual dependency information in the token stream. We also discuss the pass structure of the incremental lexing algorithm and introduce a running example. The next three sections elaborate on the implementation and analysis of each pass. In Section 5.4 we discuss the algorithm for combining a set of textual and/or structural modifications and dynamic dependency information to discover the appropriate starting location of each section requiring lexical analysis. Section 5.5 describes incremental lexing *per se* as an algorithm that traverses each outdated region to restore the consistency of its text-to-token mapping. Finally, Section 5.6 discusses the post-pass that updates information recording lexical dependencies. Each section discusses the correctness and performance of the algorithms it contains. The use of incremental lexing as part of an integrated approach to incremental software development is discussed in Section 5.7.

5.2 Related Work

Previous incremental environments have largely ignored the problem of incremental lexing by restricting the language model, the editing model, or the form of lexical dependencies. All previous approaches to incremental lexing require a static bound on the length of lexical dependencies.

Several monolingual environments have included incremental lexical analysis [23, 83]. These systems only require sufficient expressiveness for a single language, and do not constitute general solutions.

The Galaxy environment [10] touches *every* token on any textual modification to the program, and is therefore not an incremental approach. The Synthesizer Generator [81] limits the user to a single outstanding edit, for which batch lexical analysis is employed to incorporate a textual modification. PSG [6] permits *either* textual or structural modifications, but not both, and halts its incorporation of textual modifications at the first error it encounters, rather than continuing its analysis. Its lexical generator, Aladin [31], produces incremental lexers that cannot use more than a single character of lookahead.

The Pan system [8] possesses truly incremental lexical analysis, in that an unlimited number of disjoint textual edits can be applied to the program between analyses, and lexing is then applied only to the out-of-date portions of the program. However, this system also limits contextual dependencies to a single token.[5]

[3] This result assumes a balanced representation for lengthy sequences (such as declaration and statement lists); Chapter 3 explores this assumption in greater detail.

[4] Dependency analysis can be super-linear in the number of extant lookaheads; we make the reasonable assumption that this parameter is bounded by a small constant in any practical description.

[5] Regular expression notation is used for the pattern language, but many of the language features of `flex` are not provided. A special-purpose syntax is

Before:

```
...;
/* check for debugging */ # if(DEBUG==1) ...
```

After:

```
...;
/* check for debugging */ if(DEBUG==1) ...
```

Figure 5.1: A sample editing scenario used as our running example. The deletion of the # character changes the meaning of the line by altering it from a preprocessor directive to a sequence of ordinary tokens in C or C++. The edit disturbs the context surrounding the preprocessor keyword `PP_IF`, requiring the incremental lexer to replace it with a normal keyword token (`IDENT`), *even though the lexeme remains unchanged*. The lexical specification is shown in Figure 5.2.

The POE environment [32] provides per-keystroke error reporting by using a lexical analyzer capable of stopping and resuming at any character. This approach does not provide additional functionality relative to our system,[6] and actually *decreases* performance, due to the additional overhead required.

Other researchers have focused on the problem of incremental *generation* of batch lexical analyzers, as opposed to incremental lexing [43, 88]. Since language specifications are long-lived and infrequently changed relative to the number and frequency of changes made to programs written in those languages, we have been more concerned with the speed of the compile cycle than the speed of the compiler-compile cycle. Nevertheless, an incremental lexical specification for a typical programming language (C) can be generated in under 3 seconds on a typical Unix workstation and dynamically loaded into a running environment in under 1/10 second, permitting the development of the lexical specification itself to be an interactive process, without resorting to incremental or lazy generation techniques.

5.3 Framework and Overview

In this section we discuss the framework of incremental lexing, including the editing model, the representation of individual tokens and the token stream, and the interface to the batch lexer. This section also introduces our running example and concludes with an overview of the pass structure of the incremental lexical analysis algorithm.

5.3.1 A Running Example

Figure 5.1 contains the original and modified program text that serves as our sample editing sequence. The code fragment shown is in the *preprocessor* language used by C and C++. The user has modified the text in order to delay the choice between normal and debugging mode to run-time. (Previously it was decided at compile-time, requiring the use of a preprocessor conditional.) The lexical description in Figure 5.2 uses `flex`'s notation [75] to specify a portion of the tokenization required by the preprocessor language. No previously published work or existing environments supporting incremental lexing can correctly implement this combination of language features and editing sequence. Our approach not only supports this transformation, but does so optimally.

5.3.2 Token Representation

Incremental lexing is the incremental maintenance of the mapping between a text (character) stream and a token stream. Each character belongs to exactly one token, and the lexer must partition the textual stream by locating the inter-token boundaries and assigning a type to each resulting lexeme. We will assume that the mapping from a token to its lexeme is explicit and computable in constant time. Tokens persist until deleted explicitly through editing or implicitly when an invocation of incremental analysis fails to retain them in the resulting token stream. Tokens are created by the batch lexer; if the editor explicitly constructs tokens, it must ensure the correctness of their fields or consider them as outdated portions of the stream. The sentinel nodes `bos` and `eos`(Figure 3.2) are treated as the first and last elements, respectively, of the token stream. Figure 5.3 summarizes the internal representation of a token; the incremental fields are described below.

available for describing nested comment conventions (which would otherwise be inexpressible). Our approach supports non-regular patterns in a general fashion, by permitting arbitrary code to be used in the construction of tokens.

[6]While it is not our belief that character-by-character analysis is beneficial to the user, our approach can be used to implement this policy simply by requesting consistency restoration following every keystroke.

Attach symbolic names to regular expressions.

```
whitespace      [ \t]*
comment         "/*"([^*]|"*"[^/])*"*/"
ident           [_a-zA-Z][_a-zA-Z0-9]*
intconst        [1-9][0-9]*
```

Declare each user-defined state.

```
%start pp_directive
%%
```

Patterns: *Rules:*

```
{comment}                      return CMNT();
{whitespace}                   return WS();
\n                             BEGIN(INITIAL); return WS();
^/({whitespace}|{comment})*#   BEGIN(pp_directive);
<pp_directive>if               return PP_IF();
"#"                            return PND();
"("                            return LP();
{ident}                        return IDENT();
"=="                           return EQEQ();
{intconst}                     return INTCONST();
")"                            return RP();
```

Collect contiguous, otherwise-unmatched text into an error token.

```
.                              error();
%%
```

Figure 5.2: A partial lexical specification for our running example, using the notation of `flex`. The specification establishes a set of *patterns*, composed of regular expressions for identifying lexemes, and corresponding *rules* that indicate which token class to construct for each pattern. (Rules can also change the current state using the `BEGIN` directive.) The notation `<state>` restricts a rule's applicability to the named state. `INITIAL` refers to the machine's normal state. `/` indicates trailing (right) context that is required to match the pattern, but that is not part of the lexeme. A caret at the beginning of a pattern indicates that the pattern is only active when preceded by a newline or `bos`.

```
struct TOKEN {
  int type;            Token type (class).
  STRING lexeme;       Token's text.
  NODE *parent;        Parent in tree.
  Fields used and maintained by the incremental run-time mechanism:
  int state;           State of lexing machine when the token is constructed.
  int lookahead;       Number of characters read beyond lexeme.
  int lookback;        Earliest preceding token whose lookahead reached lexeme.
};
```

Figure 5.3: Representation of tokens. The `type` field is set once, when the token is constructed. The `parent` field is used by the editor to maintain a balanced tree representation. The `lexeme` and `state` fields are set when the token is constructed and maintained thereafter by the incremental run-time service, which also maintains the dependency-related fields. In practice, the incremental fields are represented implicitly, but for clarity we include them here.

Functions exported by the batch lexer:
```
int get_state ()
void set_state (STATE state)
list of TOKEN* more_tokens ()
```

Functions called by the batch lexer:
```
int next_char ()
```

Figure 5.4: Interface to the batch lexing machine. The primary connection is through the function `more_tokens`, which constructs an atomic token sequence (Section 5.3.5) and returns it. Two new operations are required of the batch machine when it is used in an incremental setting: `get_state` and `set_state`. These have no counterpart in batch analysis; they are used to record the current state when a token is constructed and to place the batch machine into a preserved inter-token state in order to re-start it at a location other than the beginning of the token stream. The `next_char` function is provided to the batch lexer by the incremental run-time service: it uses the lexemes of the persistent token stream to provide the input, as opposed to the buffering and file I/O used in a conventional batch setting.

5.3.3 Lexical Analysis Model

Explicit editing of both text *and* tree structure is permitted, with an unlimited number of edit sites and arbitrary timing in the application of lexical analysis. Both textual and structural editing will, in general, temporarily violate the consistency between tokens and their lexemes; invoking the incremental lexer will restore consistency.[7]

Since the editing model is fully general, the exact form of various operations is immaterial. A textual insertion between two tokens, for example, can be recorded as an insertion to the end of the earlier lexeme,[8] as an insertion to the beginning of the latter lexeme, or even as a structural operation that introduces a new 'placeholder' token between them. (Deleting the entire lexeme of a token offers a similar variety of representation choices.)

The incremental lexing algorithms as we present them make use of the ability to operate on multiple versions of the program simultaneously (Chapter 4). The order of the characters between the previous and current versions remains unchanged, but the token types and boundaries will, in general, be different. The lexemes of the previous version supply the input to the batch lexer. The regions requiring re-analysis are determined by the textual and structural edits applied since the reference version (along with additional areas dependent upon them, as discussed below).

5.3.4 Batch Lexer Model

The model of the batch lexer is represented by the interface in Figure 5.4. The incremental run-time system uses the batch lexer as a subroutine, calling `more_tokens` to produce the next token sequence. The incremental lexer provides the batch lexer with the `next_char` function to read characters from the lexemes of the previous token stream.

Reusing a batch lexer in an incremental environment requires addressing two additional issues: the ability to restart the batch machine at a point other than the beginning of the character stream (when it is in the `INITIAL` state) and the contextual dependencies that result when the batch lexer calls `next_char` to read characters beyond a token's own lexeme.

5.3.5 Preserving Lexing States

The `state` field in each token is used to preserve a snapshot of the internal configuration of the batch lexing automaton. At the conclusion of a rule that constructs a token, the constructor will call `get_state` to read the state of the lexer and preserve it in the token. At some later time, this preserved state information will permit the incremental run-time service to restart analysis immediately to the right of this token by passing the saved state to the `set_state` function.

Our approach does not specify the form of the state information, but does assume that it is small enough to be conveniently recorded in each token and that it can be passed to/from the batch lexer in constant time using the functions of Figure 5.4. `flex` and `lex`, for example, both require only a small integer to record an inter-token state (referred to as the 'start state' in those generators). Since any user-defined states have already been incorporated into the start state by the generator, no additional run-time mechanism is required to support this feature of description languages.[9] The beginning state (called `INITIAL` in `flex`) is stored as `bos`'s state.

[7] Note that *consistency* is distinct from *correctness*: a consistent program may contain textual sequences not permitted by the language definition ('errors'). Section 5.7.3 discusses the representation and handling of lexical errors.

[8] This is the representation used in *Ensemble*. Insertions at the beginning of the program are added to the 'lexeme' of `bos`.

[9] One feature provided by some lexical description languages is the ability to define patterns exclusive to a particular user-defined state. If a particular pattern can occur in multiple, exclusive states and the state no longer matches, a token must be re-created in order to re-label its state. While this behavior is optimal with respect to the description (and the techniques of Section 5.7.1 can restore the original token), improved incremental performance would be obtained by modifying the description to create an equivalence class for the patterns common to multiple states.

In our example, there are two states, one for handling preprocessor directive lines and another (the normal state) for processing tokens in the base language. The presence of a '`#`' character after a newline with only whitespace or comments between them signals the start of a preprocessor directive. Once in the preprocessor directive state, another newline signals the shift back to the normal state.

The relationship between patterns in the lexical description and lexemes is not necessarily straightforward. Multiple pattern matches (and their corresponding rule invocations) may be required to construct a single token, in which case only the state at the conclusion of the final rule needs to be preserved. Multiple tokens may also be constructed from the text matching a single pattern, in which case all the tokens involved are returned simultaneously from `more_tokens`. The alternative, returning a single token for each invocation of the batch lexer, requires an additional interface function to inform the incremental service when the batch lexer has reached a consistent stopping point.

Arbitrary code may be used during the construction of a single token. This seamless integration between generated analysis and user-supplied procedures is useful in constructing tokens that have non-regular syntax, such as nested comments, or for complex patterns where standard library code exists, such as floating point constants. Rules that construct (pieces of) *different* tokens may not communicate except by contextual dependencies that are reflected in the lexical description.[10] Doing so would introduce additional dependencies not visible to the incremental run-time service, and would thus result in incorrect incremental behavior.

This restriction can be relaxed by permitting a contiguous *sequence* of tokens to be treated as a single entity for the purposes of incrementality. The incremental lexer will be unable to restart analysis within such a sequence and it will require the batch lexer to reconstruct it (when necessary) in its entirety. However, as long as such regions are kept reasonably short, this represents a useful technique for constructing contiguous token sequences using unrestricted techniques without a significant loss of incremental performance. Each token in such a sequence except the final one will possess a special state value that indicates it is not a valid starting position.

5.3.6 Computing and Using Lookback Counts

To construct a token, the lexer must scan at least the characters of that token's own lexeme. In some cases, this is sufficient—parentheses are a simple example where the text of the token is sufficient to determine its right boundary. In many cases, however, at least one additional character beyond the end of the lexeme must be examined to detect that the lexeme is complete. For example, identifiers in most programming languages can be of arbitrary length, and the lexer can only be certain that an identifier is complete when it encounters a character not in the legal set of identifier spellings. In general it is not even possible to place a compile-time limit on the number of characters (or even the resulting number of tokens) of lookahead involved; doing so will artificially restrict the set of languages that can be described or the instances of those languages that will exhibit correct incremental behavior.

In a batch environment, the need to support lookahead affects buffering and I/O operations but not token construction. In an incremental setting with a persistent token stream, lookaheads must be preserved within tokens, because this information helps to provide the link between modifications made by the user and the set of tokens that require re-analysis when the incremental lexer is next invoked. Previous approaches have all used a simple, restricted scheme: a token's lookahead set is required to lie within its own lexeme and the characters of the following token. This restriction results in a trivial relationship between modifications and re-analysis: each modified token *and each token that precedes a modified token* requires re-analysis.

While the one-token dependency assumption is valid for many languages, it is not always sufficient, as our running example demonstrates: the lookahead of the newline token spans several lexemes. Our approach supports arbitrary lookahead by computing and maintaining dynamic dependency information. Contextual dependency information is stored within the tokens themselves, using the `lookahead` and `lookback` fields shown in Figure 5.3.

The lookahead is computed by monitoring the batch lexer's calls to `next_char` and the length of the resulting tokens it produces.[11] The batch machine itself does not need to be modified in any way. The *character read set* is the number of characters read by the batch lexing machine during the construction of a token; by subtracting the length of its own lexeme, we derive the *character lookahead*—whenever a character in this range is disturbed through textual editing (insertions, deletions, or overwrites) or structural editing, the token just constructed must be re-analyzed, since it might not be constructed in the same fashion given the possibly changed text to the right of its lexeme. Figure 5.5 illustrates how the incremental run-time system computes a token's lookahead set from the construction and read locations in the previous version of the token stream.

[10] Either *left context*, in the form of a specific state, or *right context*, in the form of lookahead as discussed in the next section. Long-range dependencies such as name binding [45] are outside the purview of incremental lexing, and we will never mean this type of relationship when we use the term 'contextual dependency'.

[11] Some generators require a special option to indicate that minimal lookahead is to be used. Such flags should be used to generate the best possible performance in an incremental setting.

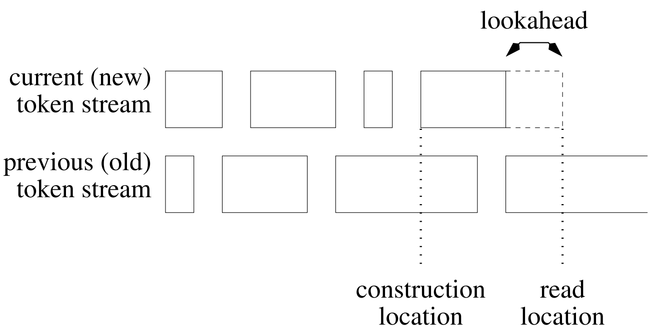


Figure 5.5: Extracting lookahead information from the batch lexer. By monitoring the difference between the *construction location* (the start of a re-lexed token) and the *read location* (the rightmost character examined by the batch lexer through a call to `next_char`), the incremental run-time system can determine the length of the character read set. The lookahead is the difference between this value and the length of the token. (Because the token boundaries in the new stream are not necessarily related to those in the previous stream, locations are <*token, offset*> pairs.)

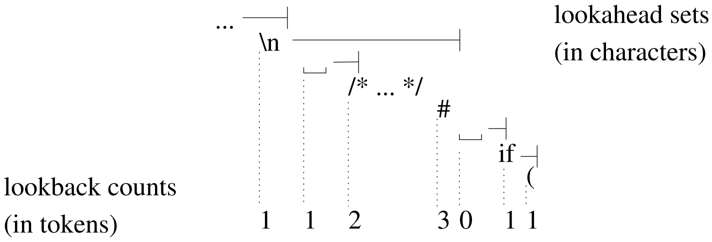


Figure 5.6: The relation between the characters read to produce a token and the resulting token lookback counts. Lookahead sets are shown in the figure as horizontal lines. During the dependency update phase, lookahead sets are converted into token lookback counts, shown at the bottom of the figure.

Token *lookback* counts are used to summarize and invert the information contained in the character lookaheads of previous tokens. Token lookback counts represent the previous tokens that are dependent on one or more characters of a given token: if a token t has a lookahead that reaches y, then y's lookback is sufficient to reach t. Lookback counts are necessary as well as sufficient: no lookback count can be reduced without violating correctness. Figure 5.6 demonstrates the relationship between lexemes, lookahead sets, and lookback counts in our running example.

The advantage of permitting unbounded token lookahead is the ability to reuse natural lexical descriptions in an incremental setting. However, most token lookahead and lookback counts in a program written in a conventional programming language will be zero or one. We can choose conservative implicit values for the character lookahead and token lookback values that cover the vast majority of tokens. This choice reduces the space requirement to that of a fixed dependency approach without loss of performance—at worst we will re-lex a fixed number of additional tokens for each modified token sequence. (The techniques of Section 5.7.1 can prevent loss of conceptually retained tokens in this case, so there is truly no penalty for the implicit representation.) The exceptional cases can be represented via an associative data structure. In the case that inter-token states are predominantly a single value, a similar technique may used to effectively eliminate the entire space cost of incrementality.

The computation of lookback values requires special handling for `eos`. Even though it possesses no explicit lexeme, the detection that no further text is present represents a type of lookahead information. Thus, we treat `eos` as if it contained a single character; any preceding tokens which read and detect the end-of-stream condition include this pseudo-character in their lookahead sets. The translation to a lookback count is then treated uniformly by the algorithm in Section 5.6.

5.3.7 Overview of the Algorithm

Incremental lexing begins with a tree where all the nested and local changes have been identified; together these form a set of paths defining an embedded tree structure within the larger tree. There are three main stages to the analysis. In the first stage (marking), the dynamic dependencies of the previous token stream are combined with the embedded change tree and used to discover the *prefix set*, the set of tokens that begin each contiguous region requiring re-lexing. Having expanded the embedded change tree to incorporate this lookback set, the second stage (lexing) then traverses each out-of-date region until the new and old token sequences once again coincide. In the third stage (dependency updating) the embedded change

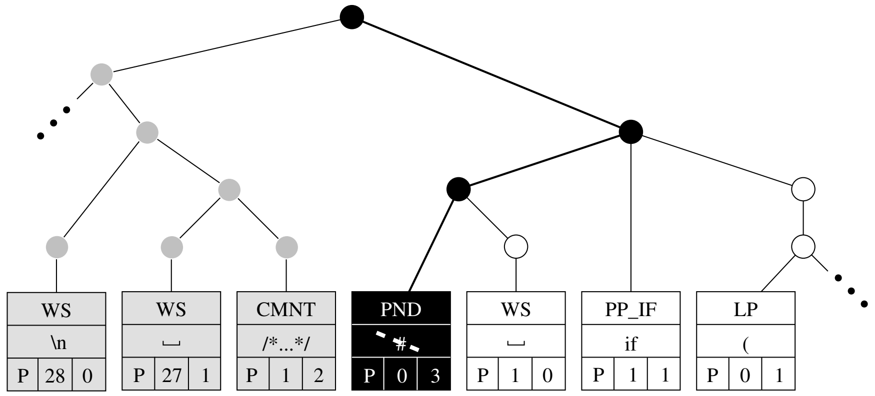


Figure 5.7: The effect of a textual edit on incremental lexical analysis. This figure illustrates the marking process when the # character is deleted in our running example. The bottom row in each token contains the preserved state ('P' denotes the `pp_directive` state), lookahead count, and lookback count while marking is in progress. The token containing the modification and the path from it to the root of the tree are shown in black. The lookback count in the modified `PND` token is used by the marking routine to discover the set of tokens affected by this change; these tokens and the additional interior tree nodes required to locate them are shown in gray.

tree (updated once again from the results of the second stage) is traversed a final time to update dynamic dependencies for each token created by phase two and any unchanged tokens that were examined. (If the range of lexical dependencies is fixed to an *a priori* value, the final stage is simply omitted.)

The three passes are conceptually distinct, although they could be applied simultaneously during a single traversal of the tree. *Ensemble* uses explicit passes both for simplicity of implementation and because a single-pass implementation would greatly complicate the interaction with incremental parsing. Separating the passes does not degrade asymptotic or practical performance.

Our incremental lexing algorithm can be used as a subroutine of incremental parsing, producing a single token (or atomic token sequence) on each call.[12] The client interface to the incremental lexer consists of the subroutines in Figure 5.11, which support starting and stopping on a region-by-region basis. (Normally, each invocation of `first_new_token` (other than the first) is for a location to the right of the previous re-lexed token, but regions guaranteed not to overlap may be processed in any order.)

5.4 Marking Phase

In order to re-lex the token stream efficiently, we need to know the starting point of each outdated region. These regions comprise the tokens that have received direct modifications, either through textual edits or by modification of the tree structure. The affected regions also include any tokens that are lexically dependent on one or more modified tokens.

5.4.1 Effect of Textual and Structural Editing on Dependencies

It is easy to see how a textual edit affects token dependencies: if a character is inserted to, deleted from, or overwritten within a token, then the token's `lookback` field gives the number of preceding tokens that must be considered suspect. Figure 5.7 illustrates the particular case of our running example.[13] No interaction with the incremental lexer is required at the time of the edit; when lexing is next requested, the current structure of the tree is used, in conjunction with the lookback counts computed when the reference version was constructed, to determine the extent of the effect.

Structural edits are more complex to handle (and also complicate the situation for text editing by potentially re-arranging the order of tokens). While it is possible to treat a structural edit as a textual edit that modifies every token in the yield of the subtree, doing so would require $O(N)$ time to analyze a subtree containing N tokens. Instead, we perform a precise computation of the possible effect of the change through history-based dependency analysis.

[12] As described in Chapter 4, the update operations of the lexer and parser occur to a logically separate tree from the version being read; thus both tools are able to traverse the previous structure of the program even though the current version of the program structure is invalid during the re-analysis.

[13] Slightly greater theoretical precision can be achieved by also using the `lookahead` fields: not all the tokens in the lookback set necessarily reach the modified token, and those that do reach it typically depend only on its left edge. However, this level of precision does not improve the asymptotic or practical performance, and requires knowledge of intra-lexeme modification sites.

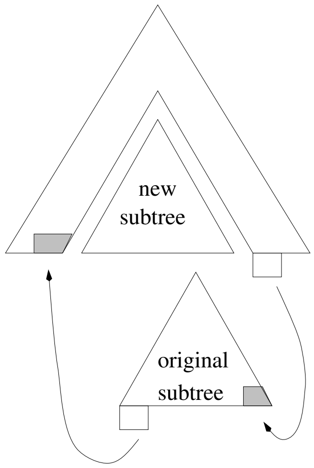


Figure 5.8: The effect of subtree replacement on incremental lexical analysis. For each replaced subtree, the lookback count in its leading token is used to determine the set of tokens affected by the edit. Both the structural traversal and the dependency analysis are with respect to the reference version. The same analysis is done for the token immediately following the replaced subtree, in order to include tokens within the subtree that are now out-of-date. (For insertions, the first step can be skipped; for deletions the latter step.)

Each structural operation is treated as a replacement; insertions replace a sentinel (completing production) with new material and deletions replace a subtree with a sentinel. The dependency analysis for a structural edit is similar to a modification of the first character in the subtree being removed and the first character following the subtree. More specifically, for each replacement point, the first token in the subtree (if one exists and if it existed in the reference version of the tree) is used to invalidate tokens with lookahead sets that reached it. The tokens affected are determined by traversing the *reference* token stream, *not* the current one. The token following the subtree replacement point is treated similarly. No other tokens can be affected by this subtree replacement, since their lookahead sets were not disturbed by it. The effect of structural editing is shown schematically in Figure 5.8.

5.4.2 Algorithm

Marking is the process of discovering the *prefix set*, the set of tokens that prefix each region requiring action by the incremental lexer. The input to this phase is a tree where all tokens and internal nodes modified since the last analysis are marked. (We refer to the modified nodes as 'implicitly' marked and the additional tokens discovered by this phase as 'explicitly' marked.) In order to locate the modified areas of the tree efficiently, each interior node on a path to one or more modified nodes is identified as possessing *nested changes*; additional nested changes are added as needed for later passes to locate the explicitly marked tokens.

The driver for the marking phase traverses an optimal path through the tree that reaches each edited site. For internal nodes (structural modifications), it first locates the tokens that may have been affected by the subtree replacement. Each implicitly marked token is then passed to a marking routine that discovers additional tokens dependent upon the changed material by using the dynamic dependency information in the `lookback` fields of the modified tokens.

Since the `state` field of each token records the batch lexer's internal state at the *completion* of a rule, the incremental lexer will pass the state saved in the token *before* the first affected token in each region to the batch lexer's `set_state` function. One complication is that only startable tokens (the final token of each sequence returned by `more_tokens`) can serve as valid starting points. For each marked token, we must therefore step backwards in the *current* version of the tree until we find a node whose `state` field indicates a valid point for re-initializing the batch lexing machine. (In practice, this is typically the previous node.) Because the nodes marked by `mark_from` are not necessarily contiguous in the current tree, `ensure_startable` must be applied to *each* marked node. Figure 5.9 contains the entry point to the marking phase.

The following theorem demonstrates the correctness of the marking phase (the extension to multiple-token sequences is straightforward).

```
Locate all the edit sites within node.
Call mark_from() on each edited terminal and the boundaries of each structural edit.
void apply_marking (NODE *node) {
  if (is_token(node) && node→text_changes(reference_version))
    mark_from(node); Handle textual changes.
  else {
    Handle structural changes.
    if (node→child_changes(reference_version))
      for (int i = 0; i < node→arity; i++) {
        NODE *old_child = node→child(i, reference);
        if (old_child != node→child(i)) {
          Mark first token not earlier than the leading edge of the original subtree.
          mark_from(first_token(old_child, reference_version));
          Mark first token after the original subtree.
          mark_from(first_token_after(old_child, reference_version));
        }
      }
    Recursively process any edits within this subtree.
    if (node→has_changes(reference_version, nested))
      for (int i = 0; i < node→arity; i++)
        apply_marking(node→child(i));
  }
}
```

Figure 5.9: Driver routine for marking algorithm. This routine locates all modifications (both textual and structural) applied since the previous invocation of lexical analysis. `first_token` returns the first token in the yield of its argument concatenated with the remainder of the token stream. `first_token_after` is similar, but returns instead the first token after the yield of its argument node. All functions that access structure have an optional argument to specify the version of the tree used for the query.

```
Explicitly mark tokens dependent upon tok for re-lexing.
This backup occurs in the reference version.
void mark_from (TOKEN *tok) {
  if (!tok→exists() || !tok→exists(reference_version)) return;
  ensure_startable(tok);
  Check everything in its lookback set.
  for (int ov = tok→lookback; ov > 0; --ov) {
    tok = previous_token(tok, reference_version);
    if (tok == bos) return;
    if (!tok→exists() || marked(tok)) continue;
    mark(tok);
    ensure_startable(tok);
  }
}

Ensure that we have a valid state to re-start the lexer here.
This backup occurs in the current tree.
void ensure_startable (TOKEN *tok) {
  for (TOKEN *tok2 = previous_token(tok);
       !startable_state(tok2) && !marked(tok2);
       tok2 = previous_token(tok2)) mark(tok2);
}
```

Figure 5.10: Marking algorithm. All tokens from the reference version that are still present in the token stream and that read one or more characters in `tok`'s lexeme are explicitly marked. In addition, the algorithm ensures that the batch lexer can be restarted at the beginning of each outdated region by calling `ensure_startable` for both `tok` and any explicitly marked tokens.

Theorem 5.4.2.1
The marking algorithm marks all and only those tokens requiring re-analysis that are not themselves modified.

Proof To see that this test is sufficient, suppose there exists a token t with at least one modified character in its lookahead set. If the character is within its own lexeme, then t is marked by the modification itself. If the character was in a different token modified through a textual edit, then the token that contained it must have had a lookback field at least large enough to encompass t, and `mark_from` would thus have marked t. The only remaining possibility is that the altered lookahead arose through a structural edit. In this case t was separated from a token containing one or more characters in its lookahead set through a subtree replacement. Without loss of generality, assume that t was to the left of the original replaced subtree in the reference version. Since t's lookahead set extends into the left edge of the subtree, the marking algorithm must have included t in the set of tokens marked when this structural edit point was processed. Hence t is actually marked.

To see that the test is also necessary, we merely observe that any explicitly marked token had at least one character in its lookahead set modified through one or more editing operations since the previous lexical analysis.

With regard to running time, marking *per se* examines only old tokens containing affected characters. It is thus linear in the number of affected characters and affected tokens and clearly optimal.

5.5 Lexing Phase

Lexing is the process of repairing a contiguous region of affected characters by reading the (possibly changed) lexemes from the previous token stream and invoking the batch lexing machine to re-create that portion of the new token stream. Lexing is applied to each outdated region in turn, beginning with the next token in the prefix set not yet visited. In order to stop lexing a region, we must ensure that the construction location (Figure 5.5) is at the beginning of an unmarked token and that the last newly lexed token contains a startable state matching the state in the previous token of the previous stream.

The routines comprising the lexing pass are shown in Figure 5.11. These routines can also be called directly by an incremental parser. Figure 5.12 illustrates the use of these routines by implementing the lexing phase as a standalone operation.

Theorem 5.5.1
At the conclusion of the lexing phase, the token stream is identical to the token stream that would result from executing the same batch machine on the concatenation of the lexemes. Furthermore, each token records the state of the batch lexing machine at the point of the token's construction.

Proof The correctness of the marking phase and batch lexer are assumed. We proceed by induction over the token stream as it exists immediately prior to the start of the lexing phase. The base case is simple: the beginning of stream markers and initial lexical states are clearly the same in both the batch and the incremental streams.

For the inductive case, assume that the lexemes of the preceding $N - 1$ tokens have been correctly lexed and that the internal state of the batch lexer is the same in both cases. If the current token is unmarked, then the state of both machines is equivalent, the characters of the token's lexeme and lookahead set are unchanged from the previous invocation, and thus the old token may be safely reused (and the state with which it is labeled corresponds to the state of the batch machine at the conclusion of the rule creating the token).

If the current token is marked, then the batch lexers will read the same set of characters in the same state, and thus produce the same stream of tokens until the stopping condition obtains. At this point, the next character to be consumed and the state of the batch lexing machine correspond to the 'leading edge' of an unmarked token in the old stream, completing the inductive step.

This phase touches only old tokens that are marked, for which the starting state or offset has been changed, or that are part of a previous atomic sequence. The number of invocations of the batch lexer (and number of tokens examined in the new stream) is therefore optimal and is clearly linear in the total number of affected characters.

5.6 Lookback Update Phase

When the lexing phase completes, all the fields in each token in the stream are correct with the exception of the lookback counts. The `lookback` field will be undefined for each token produced by either `first_token` or `next_token`. In addition, newly constructed tokens may have read characters from lexemes in unchanged tokens. (Conversely, they may have failed to read as far as previous analyses did into the unchanged region.) The lookback phase handles both types of updates.[14]

[14]It *is* possible to perform this update in parallel with the lexing phase, but doing so greatly complicates the algorithm. When lexing is performed in parallel with incremental parsing, on-line dependency updating is even more difficult, since the new tree structure is fragmented until parsing completes.

Begin incrementally lexing a new region starting at `tok`.

```
TOKEN *first_new_token (TOKEN *tok) {
  read_token = construction_token = tok;
  read_location denotes <read_token, read_offset>;
  construction_offset = read_offset = 0;
  construction_location denotes <construction_token, construction_offset>;
  if (tok == bos) batch_lexer→set_state(INITIAL_STATE);
  else batch_lexer→set_state(previous_token(tok)→state);
  token_list = ∅;
  return next_new_token();
}
```

Return the next re-lexed token.

```
TOKEN *next_new_token () {
  if (token_list == ∅) token_list = batch_lexer→more_tokens();
  for each tok in token_list {
    if (tok is last element) tok→state = batch_lexer→get_state();
    else tok→state = unstartable_state;
    advance(construction_location, tok→length);
    tok→lookahead = delta_in_chars(read_location, construction_location);
  }
  return last_token = remove first token in token_list;
}
```

Determine when previous and current token streams merge again.

```
bool can_stop_lexing () {
  return
    token_list == ∅ && construction_location.offset == 0 &&
    !marked(construction_location.token) && is_startable(last_token→state) &&
    last_token→state ==
      previous_token(construction_location.token, previous)→state;
}
```

Incremental run-time service provides this to batch lexer to read from
lexemes in the previous version of the token stream.

```
int next_char () {
  while (read_offset == read_token→length && read_token != eos) {
    read_token = next_token(read_token, previous_version);
    read_offset = 0;
  }
  if (read_token == eos) return -1;
  return read_token→lexeme[read_offset++];
}
```

Figure 5.11: Lexing algorithm. The input to this algorithm is a marked token stream. The output is a (possibly changed) token stream that is identical to one produced by executing the batch lexer on the concatenation of the previous stream's lexemes. The `next_subtree` function returns the node following its argument's rightmost descendant in a DFS ordering.

Restore consistency to the entire token stream.
(Operations to incorporate tokens into the tree structure are not shown.)

```
void lex_phase () {
  for (TOKEN *tok = find_next_region(root);
       tok != eos;
       tok = find_next_region(tok)) {
    tok = first_new_token(tok);
    while (!can_stop_lexing()) tok = next_new_token(tok);
  }
}
```

Find the next marked token within or after `node`.

```
TOKEN *find_next_region (NODE *node) {
  if (node == eos || (is_token(node) && marked(node))) return (TOKEN*)node;
  if (node→has_changes(nested)) return find_next_region(node→child(0));
  return find_next_region(next_subtree(node));
}
```

Figure 5.12: Driver routine for the lexing phase, when used in a standalone fashion. Incremental lexing can be intermixed with parsing by having the incremental parser call the routines in Figure 5.11 directly.

Region	Token	Lexeme	Length	Lookahead list
re-lexed	WS$_1$	\n	1	<WS$_1$,28,0>
	WS$_2$	␣	1	<WS$_1$,27,1> <WS$_2$,1,0>
	CMNT	/*...*/	25	<WS$_1$,2,2>
	WS$_3$	␣	1	<WS$_1$,1,3> <WS$_3$,1,0>
	IDENT	if	2	<IDENT,1,0>
synching	LPAREN	(	1	<LPAREN,0,0>

Figure 5.13: Updating lookback counts in a section of the token stream. The bootstrap section is empty in our running example, so processing begins with the first re-lexed token (the newline). We maintain the invariant that the list of lookahead sets contains all and only the lookaheads that reach the lexeme of the current token. When we finish processing lookaheads for re-lexed tokens and find a match between the computed and stored lookback counts, the region is complete. In the example, this occurs when the left parenthesis is encountered.

5.6.1 Algorithm

During lexing, the character lookahead set for each token is preserved in its `lookahead` field at the time the token is constructed. The lookback update phase consists of transforming these character lookahead counts into token lookback counts. The algorithm's central data structure is a *character lookahead list* that keeps track of multiple outstanding lookaheads. As each token is processed, its character lookahead is added to this list and any lookaheads that terminate within this token's lexeme are removed from the list.

The lookahead sets for our running example are shown graphically in Figure 5.6. When the comment token is processed, one lookahead is removed (the preceding whitespace token) and one remains (the newline token). The comment token itself has no lookahead, so no entries are added. Figure 5.13 shows the lookahead list immediately after processing each token.

Each region is processed in three parts: a bootstrap section, a re-lexed section, and a synchronization section. For the middle section, composed of the re-lexed tokens, the algorithm computes the lookback count for the token based on the contents of the lookahead list. Then each lookahead set in the list is advanced by the length of the token's lexeme, and the token's own lookahead is added.

In order to maintain the invariant that the lookahead list contains all and only lookahead sets from the current version of the token stream that reach the token being processed, we may need to initialize the lookahead list from lookaheads in tokens that *precede* the first re-lexed token. To determine which tokens are included in the bootstrap section, we first note that the token preceding the first (re-lexed) token in the region must exist in the current, previous, and reference version of the token stream. The token that follows it in the reference version contains the relevant lookback count, and we continue adding preceding tokens to the bootstrap section until this count is exhausted or until we discover a point where the token streams differ (indicating that tokens to the left have already been processed).

The symmetric problem arises following the re-lexed tokens: because their lookahead sets may have penetrated (or now fail to penetrate) unchanged tokens that follow, we must continue updating lookbacks until we reach `eos` or the next re-lexed token or until we meet two conditions simultaneously: the lookback list contains no elements from re-lexed tokens

Find and update each modified region of tokens.
```
void update_lookbacks () {
  NODE *node = root;
  while (node)
    if (is_token(node)) {
      TOKEN *tok = (TOKEN*)node;
      if (was_re_lexed(tok)) node = fix_lookbacks(tok);
      else node = next_subtree(node);
    } else if (node→has_changes(nested)) node = node→child(0);
    else node = next_subtree(node);
}
```

Figure 5.14: Driver routine for lookback recomputation.

(which would imply that we haven't finished updating all the relevant lookback counts) *and* the lookback computed from the lookahead list is the same as that stored in the token being processed. The latter condition is necessary to handle shrinking lookaheads from one version of the token stream to the next.

Figure 5.14 contains the driver that locates each region requiring lookback processing. The actual updating is performed by the `fix_lookbacks` routine, shown in Figure 5.15.

Theorem 5.6.1.1
At the conclusion of the lookback update phase, each token t in the new stream has a lookback value b such that the earliest token in the stream with a lookahead extending into t is the b^{th} previous token.

Proof Consider the tokens processed by a call to `fix_lookbacks`. The search for the left edge of the bootstrap section is terminated either by the earliest token whose lookahead penetrates the current token, by a token that was re-lexed, or by `bos`. In the first case, the local token stream is the same in the current, previous, and reference versions, so the lookback count itemized all and only the tokens with relevant lookaheads. In the latter case, the re-lexed token's lookahead has already been processed in full by induction, so again the lookahead list contains all and only the relevant lookahead sets.[15]

We assume the correctness of the lookahead list operations; the lookback count assigned to each re-lexed token is therefore necessary and sufficient by the invariant that the lookahead list contains all and only the lookahead sets reaching the current token. The processing of each re-lexed token clearly maintains that invariant.

For unchanged tokens to the right of the re-lexed section that include characters read by one or more re-lexed tokens, the invariant on the lookahead list's contents remains unchanged. We now examine the stopping condition. The cases of encountering a re-lexed token or `eos` are trivially correct. The remaining case requires two conditions to hold simultaneously: the lookahead list consists entirely of tokens that have *not* been re-lexed, and the lookback count to be assigned to the current token matches its stored value. The conjunction is clearly sufficient, since the lookback count computed for the next token would necessarily match the value of its `lookback` field. The test is also necessary: it is straightforward to exhibit a counter-example to demonstrate that a violation of either condition results in insufficient or over-estimated lookback counts.

When the outer loop in `fix_lookbacks` terminates, the token returned possesses a lookback count that is necessary, sufficient, and unchanged from its reference value (or is `eos`). The *unchanged* token sequence terminated by the next call to `fix_lookbacks` therefore possesses the same property.

Corollary 5.6.1.2 *Lookback processing examines the minimal number of tokens.*

Lookback processing is clearly linear in the number of affected tokens and characters; the overhead of tree traversal is the same as in the previous phases.

5.7 Incremental Lexical Analysis in an ISDE

Batch lexical analysis is useful in a number of situations, and incrementality is applicable to several of them. Our primary interest, however, is the use of incremental lexical analysis as a component of an ISDE. In this case the token stream is part of the persistent (structural) program representation, and an incremental parsing algorithm is the 'client' of the incremental lexer.

[15]It is true that not all of the tokens in the bootstrap section necessarily have lookaheads reaching the first character of the first re-lexed token in the region: as always, a lookback count represents the union of of lookahead sets, and one or more earlier tokens may have been changed. However, the bootstrap region's processing is sufficient and, given that the only tokens it enters are ones that *could* possess relevant lookaheads, necessary.

```
Process a re-lexed region starting at tok.
TOKEN *fix_lookbacks (TOKEN *tok) {
  la_set = ∅;
  if (tok != bos) {
    Extract lookback count (if different in current version, use old value).
    int lb = next_token(previous_token(tok), reference_version)→lookback;
    TOKEN *boot_tok = tok;
    while (--lb > 0 &&
          previous_token(boot_tok, reference_version) ==
            previous_token(boot_tok, previous_version) &&
          !was_re_lexed(previous_token(boot_tok)))
      boot_tok = previous_token(boot_tok);
    Initialize the lookahead set from the bootstrap region.
    while (boot_tok != tok) {
      la_set.advance(tok→length);
      la_set.add_item(tok);
      tok = next_token(tok);
    }
  }
  do {
    Set the lookback for re-lexed tokens.
    while (was_re_lexed(tok)) {
      tok→lookback = la_set.compute_lookback();
      la_set.advance(tok→length);
      la_set.add_item(tok);
      tok = next_token(tok);
    }
    Symmetric to bootstrap: process unmodified tokens reached by lookahead from re-lexed area.
    while (tok != eos && !was_re_lexed(tok) &&
           !la_set.all_items_discardable() &&
           tok→lookback != la_set.compute_lookback()) {
      tok→lookback = la_set.compute_lookback();
      la_set.advance(tok→length);
      la_set.add_item(tok);
      tok = next_token(tok);
    }
  } while (was_re_lexed(tok));
  return tok; Return first clean token or eos to caller.
}
```

Figure 5.15: Update algorithm for a contiguous range of modified tokens. The driver routine is shown in Figure 5.14. The operations on the list of lookaheads are defined in Figure 5.16.

```
advance (int offset)
  replace <tok,cla,cnt> in list with <tok,cla - offset,cnt + 1>

int compute_lookback ()
  remove <tok,cla,cnt> s.t. cla <= 0 from list
  if (list == ∅) return 0;
  else return max cnt | <tok,cla,cnt> in list

add_item (TOKEN *tok)
  add <tok,tok→lookahead,0> to list

bool all_items_discardable ()
  ∀ <tok,cla,cnt> in list, !was_re_lexed(tok)
```

Figure 5.16: Routines to update the lookahead list during lookback processing. Each entry in the list is a triple consisting of a token, a character lookahead count, and a token lookback count.

```
bool bottom_up_reuse_test (TOKEN *tok) {
  if (construction_location.token→type == tok→type &&
      construction_location.token != last_reused_token &&
      construction_location.token != eos) {
    construction_location.token→state     = tok→state;
    construction_location.token→lexeme    = tok→lexeme;
    construction_location.token→lookahead = tok→lookahead;
    Treat this token as re-lexed during lookback update phase.
    set_tok_was_re_lexed(construction_location.token, true);
    last_reused_token = construction_location.token;
    return true;
  }
  return false;
}
```

Figure 5.17: Computing bottom-up reuse during incremental lexing. `next_new_token` is modified to apply this test to all tokens passed over when updating the construction location through calls to `advance`. If an old token can be reused, the new information is copied into its fields. (Although more aggressive strategies could be employed, they are typically subsumed by top-down reuse. The simple scheme shown here captures the common cases and creates additional top-down reuse possibilities by 'seeding' the discovery process.)

The algorithms we have discussed so far can be applied without change in this setting. In this section we describe three additional topics primarily of interest within an ISDE: the preservation of information through *token reuse*, the *reversibility* of the transformation induced by incremental lexical analysis, and the issue of *error detection and recovery*.

5.7.1 Token Reuse

As described in Section 4.5, the analysis and transformational tools in an ISDE should be designed to reuse physical nodes whenever they are logically unchanged: the maintenance of associated information can be implemented most efficiently when the physical identity of an item matches its conceptual identity. Token reuse also improves performance because reuse calculation is significantly faster than additional incremental reevaluation by semantic analysis and other tools, lowering the *total* amount of work performed in response to the original program modifications.

Because our incremental lexical analysis algorithm is optimal, it intrinsically reuses that portion of the token stream provably unaffected by the user's modifications. However, *any* analysis is inherently conservative, and many common modifications result in re-lexing tokens that are conceptually unchanged, constructing new tokens isomorphic to deleted tokens. (For example, consider an editing sequence where the user 'undoes' a character insertion by deleting it, rather than reverting to the previous version of the document.)

At other times some *field* of a token that is not part of the user model is the only change; in this case the token can also be reused simply by updating the appropriate field's value. (The latter case occurs in our running example, where the `state` fields of the re-lexed tokens are altered, but the user-visible information—location, type, and lexeme—remain unchanged.)

We will consider two different approaches to reuse.[16] The first, *bottom-up reuse*, is computed directly by the incremental lexer as it operates. In our running example, it is easy to see that the re-lexed tokens can be reused: only the `state` fields will change from the previous version of the stream to the new version.

More generally, we compute bottom-up reuse by examining tokens in the previous version of the stream that would not otherwise be incorporated in the new version. When the type of a newly constructed token matches the type of a token being eliminated, the old token can be reused by copying the `lexeme`, `state`, and `lookahead` fields from the new token (the new token is then discarded). A simple heuristic for discovering reuse possibilities is shown in Figure 5.17.

Bottom-up reuse captures the reuse cases that can be easily discovered using only local information, and it is both efficient and simple to implement. Bottom-up reuse is also necessary to allow the parser to reuse interior nodes in the program structure. However, the inability of the bottom-up reuse process to consider more complex comparisons between the old and new versions of the token stream (such as out-of-order reuse) can result in missed opportunities. We therefore apply another type of reuse based on structural comparisons, referred to as *top-down reuse*.

Unlike bottom-up reuse, which is performed in parallel with incremental lexing, top-down reuse is performed as a fourth pass, after incremental lexing and parsing have completed. It involves a recursive traversal of the current document structure, limited to the modified regions. Each non-new node with one or more changed children is subject to the top-down check, which attempts to replace any new child with its counterpart from the previous version of the tree. (A more general

[16] Node reuse by the incremental parser, including a definition of optimality, is discussed in Chapter 6.

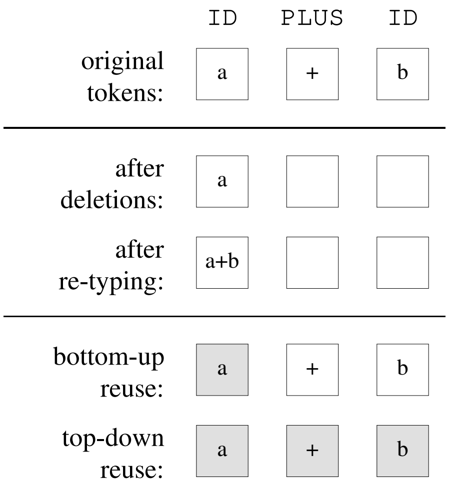


Figure 5.18: Example of token reuse. In this scenario, the user deletes the text corresponding to two lexemes, then 'reverses' his decision by re-typing the characters instead of undoing the operation (a common pattern). After re-lexing with only bottom-up reuse, the initial token in the sequence will be reused, but the remaining two tokens will be re-created; any annotations associated with them would then be lost. Applying top-down reuse results in the restoration of all three tokens. In more complicated examples, bottom-up and top-down reuse must be used in combination to achieve the best results.

discussion of top-down reuse, and the algorithm to compute it, are provided in Section 6.7.) In combination, the two types of reuse restore virtually every token that the user would consider unchanged.[17]

5.7.2 Reversibility

The transformation induced by incremental language analysis, including lexical analysis, should be considered an update to the program in the same manner as textual or structural edits. Since any user operations will be undoable, it is desirable for the language-based transformations to be undoable as well. A uniform treatment of program updates results in a more coherent and comprehensible user interface: every update can be undone using the same interface, and every transformation possesses the same semantics in the user's model. Since we use low-level history services to enable efficient analysis in the first place, it is only natural that they should record the resulting transformation in a manner that is uniform with user-supplied modifications.

To enable efficient reversibility of the incremental lexing operation, all that is required is that versions of the token stream other than the current one can be restored by the history services without violating correctness. In our representation, the history services will already be versioning the user-visible information in each token (the lexeme and the parent link—the `type` field is read-only information, and therefore is the same in every version). Reversibility requires that we also version data specific to the incremental lexer: the `state`, `lookahead`, and `lookback` fields.[18] When this is done, the history services can be used to alter the current version of the program without involving the incremental lexer.

5.7.3 Error Recovery

There is an important distinction between *inconsistency*, which is a transient state where one or more modifications have rendered the token/lexeme relationship potentially invalid within some regions, and *errors*, which indicate character sequences that are not admitted by the language definition. Our model is one where both valid and invalid editing operations are permitted, with the various analysis/transformation tools discovering the maximum amount of information in the presence of any errors that arise.[19] Errors in the program text (such as characters not in the language's accepted character set)

[17] The bottom-up mechanism allows the incremental lexer to operate in tandem with the incremental parser, but does not guarantee the maximum number of reused tokens. If incremental lexing occurs instead as a separate pass, the optimal set of reused tokens (defined as the set of tokens for which the offset range and type are unchanged between the previous and current version) can be discovered easily using a left-to-right scan of the changed regions. Various heuristics (such as retaining a majority or a subset of the original characters in the new version of the token) can be employed to retain additional tokens that represent reuse from the user's perspective. (Additional reused tokens may also be discovered by top-down reuse when such heuristics are considered.)

[18] The lookback updating stage already requires that we retain access to the last-lexed value of lookback counts until that pass has completed. As mentioned in Section 5.3.6, an explicit representation of some or all of these fields can be avoided in most cases.

[19] Another solution is to prevent erroneous modifications from being made. While we feel that restrictive, generative approaches to software development are unnecessary and undesirable, the algorithms described here can be easily applied to this editing model as well. By running the lexical analysis algorithm in a *read-only* mode, an alteration can be checked for compatibility, with the analysis algorithm returning an error indicator instead of permitting an invalid

or at the lexical level may discovered most naturally by the lexer.

In a batch environment, a lexical description often includes a simple scheme for handling characters that cannot be incorporated by other ('normal') patterns. The rule accompanying this default pattern then emits an error message. A similar solution can easily be provided in an interactive domain by defining a distinguished token type to represent unmatched text. (Either the pattern in the lexical description or the incremental run-time service should ensure that contiguous unmatched characters are always combined into a single 'unmatched text' token.) The representation of unmatched tokens in the parse tree can be handled similarly to those for explicit whitespace (Appendix C). In the lexical description for our running example (Figure 5.2), the final pattern absorbs unmatched characters.

A similar approach can be taken to programmer-supplied error patterns, which typically operate by recognizing a superset of the actual language and then distinguishing correct lexical structures from 'near misses'. A simple example would be a language that limited the length of identifiers; the rule would construct either a normal or erroneous identifier depending on the number of characters in the lexeme. Error tokens of this form are explicitly typed, are are distinct from normal tokens. No special support is required in the incremental lexer to detect or handle programmer-supplied error patterns.

A more general approach to error recovery that uses the interactive and history-based nature of the ISDE to its full advantage is described in Chapter 8.

5.8 Conclusion

The algorithms presented in this chapter constitute the first published technique for language-independent incremental lexical analysis that supports the full pattern set of conventional batch generators and runs in optimal space and time. It thus provides the maximum amount of expressiveness, enabling the lexical characteristics of real programming languages to be described in a natural manner without requiring either the language description writer or the lexical generator to compute a limit on the length of lexical dependencies. Existing lexical analyzer generators can be used without modification. The performance of our automatically generated incremental lexers rivals hand-written approaches, and the generation process itself is fast enough to enable rapid debugging and prototyping of new lexical descriptions.

In an interactive software development environment, our approach to incremental lexing retains useful information and minimizes changes through aggressive reuse computation, using both bottom-up and top-down strategies. The incremental lexer can be invoked as a separate transformation pass or as a subroutine of the incremental parser. Explicit error patterns and the efficient reversibility of the lexing transformation are supported without changes to the generator or incremental evaluator.

---

update. Alternatively, lexical analysis can be allowed to proceed normally, followed by a *post hoc* check to determine whether all transformations it induced were legal; if not, some or all of the transformations can be efficiently discarded (Table 3.2).

Chapter 6

Efficient and Flexible Incremental Parsing

Previously published algorithms for LR(k) incremental parsing are inefficient, unnecessarily restrictive, and in some cases incorrect. In this chapter 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. The self-versioning document representation of Chapter 3 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.

6.1 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 [7, 36, 49, 58, 104]. Although the topic 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 issue required 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` [18] 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 ISDE.

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 signifi-

[1] Gafter [35] is the notable exception, but his approach precludes possibly-empty sequences, which arise in virtually every programming language.

[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 [53] that can be statically encoded so that the parser remains deterministic.

cantly 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.

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 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 chapter is organized as follows. Section 6.2 compares previous work on incremental parsing to our requirements and results; it is not needed to understand the material that follows. Section 6.3 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 6.5. Section 6.6 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 6.7 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 4.3. Error *detection* is discussed in conjunction with the incremental parsing algorithm; error *recovery* is described in Chapter 8.

6.2 Related Work

Several early approaches to incremental parsing use data structures other than a persistent parse tree to achieve incrementality [3, 106]. 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 [81] or to editing of only a select set of syntactic categories [21], or can only parse up to the current (single) cursor point [86]. 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 [36]. 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 [10, 71, 86]. Li [60] describes a sentential-form LL(1) parser that can accommodate multiple edit sites.

Jalili and Gallier [49] 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 by *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 [58] 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. His work focuses on the indirect performance gains that accrue from *node reuse* in an ISDE. But unlike the original LR(0) algorithm, his 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

[4] Section 6.3 reviews state matching.

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 [76] 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 [105] recognizes the utility of sentential-form parsing, but still records parse states in nodes and thus requires a post-pass to relabel subtrees. Li [61] 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 require 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 [58] requires five extra fields per node). Only Degano et al. [21] 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 sub-optimal 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 tool `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.

6.3 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 6.4 extends these results to achieve optimal incrementality; subsequent sections discuss representation issues and additional functionality.

6.3.1 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.

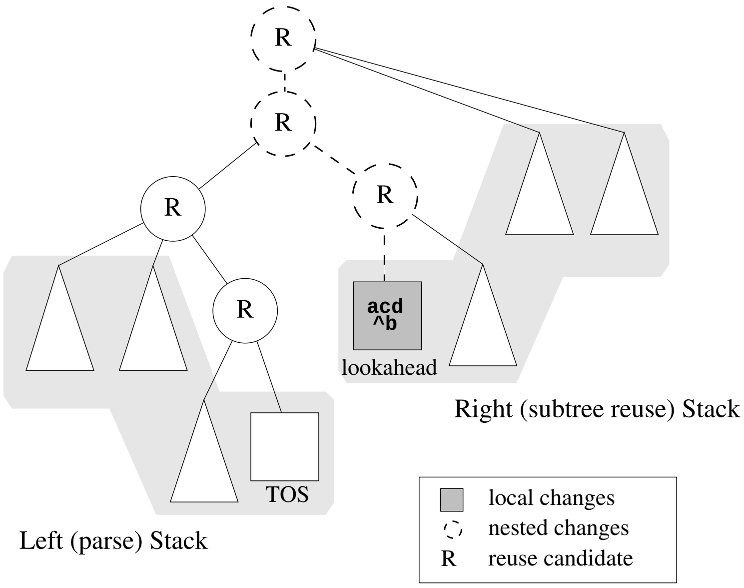


Figure 6.1: Incremental parsing example. This figure illustrates a common case: a change in the spelling of an identifier results in a 'split' of the tree from the root to the token containing the modified text. The shaded region to the left becomes the initial contents of the parse stack, which is instantiated 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 6.7). 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.

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 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,[5] 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 6.1 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 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. [5].

Theorem 6.3.1.1
Consider a conventional batch LR(1) *parser in the configuration:*

[5] All textual and structural modifications are reflected in the tree itself. Section 4.3 discusses the representation of programs and the techniques for summarizing changes.

```
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);
}
```

Figure 6.2: 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.

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

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:

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

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 6.3.1.1 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.[6]

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 6.2). 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.[7] Figure 6.3 illustrates the breakdown process. The following theorem relates the configuration of a batch parser to that of an incremental parser that has shifted a nonterminal and then invoked `right_breakdown`, by showing that the parse stack contents, parse state, and lookahead symbol are identical.

Theorem 6.3.1.2
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:*

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

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

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

[6] In Section 6.4 we describe a technique that involves minimal changes to table construction methods and provides better incremental performance than terminal lookahead information could achieve.

[7] 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 6.5.

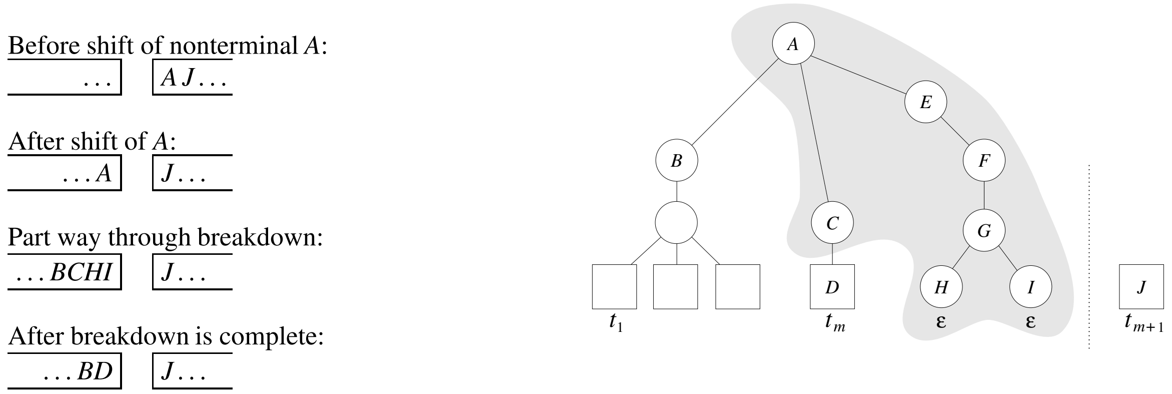


Figure 6.3: Illustration of `right_breakdown`. The shaded region shows the reductions 'undone' by the breakdown—all nodes representing reductions predicated on the following terminal symbol (J) are removed. Any subtrees with null yield are discarded, then the right-hand edge of the subtree on top of 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.)

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 6.2) 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.

6.3.2 An Incremental Parsing Algorithm

We now use Theorem 6.3.1.2 to construct an incremental parser. Figure 6.4 presents pseudocode for this algorithm.

The algorithm in Figure 6.4 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 4.3. 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 6.3.1.2, which associates the configuration of the incremental parser immediately prior to each reduction with a corresponding configuration in a batch parser.

Theorems 6.3.1.1 and 6.3.1.2 apply equally well to LALR(1) and SLR(1) parsers, so the algorithm given in Figure 6.4 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 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 6.4,[8] 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,

[8] Section 6.6 discusses the model of incremental parsing and the assumptions regarding the form of the grammar and parse tree representation.

```
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);
      }
}
```

Figure 6.4: An incremental parsing algorithm based on Theorem 6.3.1.2. 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 6.5; `has_changes` is a history-based query from Figure 3.2. `bos` and `eos` are the token sentinels illustrated in Figure 3.2.

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;
}
```

Figure 6.5: 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 6.5).

$O(\lg N)$ 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.

6.4 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 6.4 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 'next terminal' pointers in the tree nodes.

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.[9] 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

[9]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 [5] are easily modified to enumerate all lookahead symbols rather than terminals alone.

```
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);
      }
}
```

Figure 6.6: Improved incremental parsing algorithm. Its correctness is expressed by Theorem 6.4.1.1. 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 tentative 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.

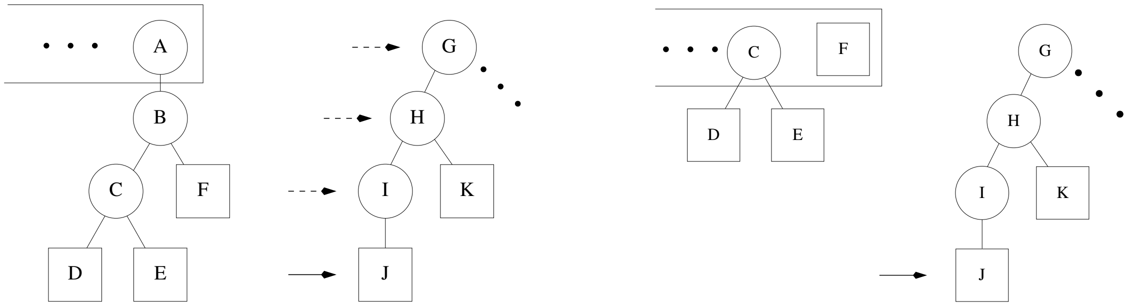
be discovered before k terminal symbols past the point of the invalid action have been shifted. The parser backtracks efficiently from invalid transitions; in the $k = 1$ case, backtracking is merely a delayed invocation of `right_breakdown`.[10] Optimistic behavior thus improves both the asymptotic and the practical performance of the incremental parser.

The algorithm in Figure 6.6 implements the optimistic strategy by a technique similar to the *trial parsing* used in batch parser error recovery [14]. Suppose we can legally shift a reused subtree, and, *in the 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 of the initial subtree, proving that any reductions optimistically retained (or applied based on insufficient lookahead) were indeed valid.[11] These reductions include any subtree 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 6.7 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 tentative (unverified) reductions. At this

[10] If the grammar contains V nonterminals, the amount of backtracking is limited to $O(kV)$.

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



7a. Discarding left edge of right stack.

7b. Discarding right edge of left stack.

Figure 6.7: 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.

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 6.6 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 the error is detected.[12] (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 validate any speculative actions, just as shifting the following terminal symbol would. Lossy compression of the terminal reduction actions and the invalid reductions permitted by other parsing classes (LALR(1), SLR(1)) limit validation to shifts of non-ϵ-subtrees.[13] 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 the algorithm in Figure 6.6 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 6.1, 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 similar to its terminal counterpart.

6.4.1 Correctness

We now demonstrate that the parse tree produced by the algorithm in Figure 6.6 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.[14]

First, we justify the optimized shifting strategy. Recall that Theorem 6.3.1.1 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 6.3.1.2 ensures that the configuration of the batch and incremental parsers are identical after the shift operation, even without the breakdown procedure.

[12] Replacing error entries in the terminal transition portion of a canonical parse table with reductions has the same effect.

[13] 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.

[14] 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.

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.
- 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 6.4.1.1
The configuration of the incremental parser defined by the algorithm in Figure 6.6 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 6.4.1.2 *The incremental parsing algorithm of Figure 6.6 produces the same parse tree constructed by a batch parser reading the same terminal yield.*

6.4.2 Optimality

We now investigate the claim that the algorithm in Figure 6.6 (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 6.7 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 6.2 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 6.2. 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.

6.5 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.[15] 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 [4].[16]

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 6.8 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.

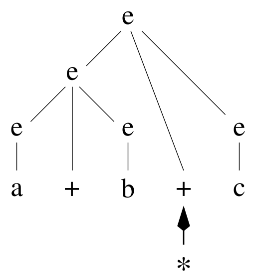
One possible solution would be to encode the dynamic selection of desired parse trees in the incremental parsing algorithm itself. For example, an existing theory of operators [1] could be extended to produce an incremental evaluator by maintaining synthesized attributes that describe, for each expression 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 specifications. This technique would be limited to the class of ambiguities addressed by the operator notation.

[15] 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. The following chapter discusses incremental parsing of languages in which ambiguity cannot be statically resolved.

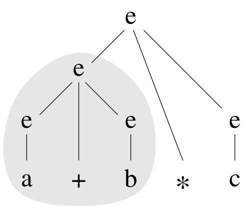
[16] These resolution methods are the most widely used, but have several theoretical disadvantages, including the fact that they may result in incomplete or even non-terminating parsers. Thorup [90] examines methods to eliminate conflicts while preserving the completeness, termination, and performance results of conventional LR parsers. Klint and Visser [53] 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 ')'
  ;
```

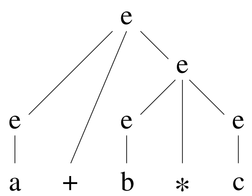
8a. Bison input file.



8b. Original program with edit site marked.



8c. Incorrect parse due to ambiguity; the reused subtree is shaded.



8d. Correct parse due to additional breakdowns.

Figure 6.8: 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).

6.5.1 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.[17]

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 6.8, 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 6.9. 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.[18]

6.5.2 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 6.5, 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 6.8) 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 matching uses less time and space than full state-matching parsers, while supporting the same class of conflict resolution mechanisms.

6.6 Representing Repetitive Structure

The asymptotic performance results presented in this chapter 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

[17] Visser [95] examines an alternative approach that instead modifies the item set construction to encode parse forest filters [42]. 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.

[18] 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;
  }
};
```

Figure 6.9: Computation of dynamic fragility.

separate statements. The former do not represent true performance problems because the sequences they construct are naturally limited; for instance, one assumes that the size of an expression tree in C is bounded in practice. The latter type are problematic, since their sequences are usually 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.

To avoid this problem, we represent associative sequences non-deterministically; the ordering of the yield is maintained, but otherwise the internal structure is unspecified [35]. 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 [89], 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 [56]. 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 to *Ensemble* (including Java and Modula-2) amounted to less than 1% of their text. These changes also simplify 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.[19] The form of the productions expresses the associativity of the sequence; Figure 6.10 illustrates the transformation.

The incremental parsing algorithm requires no changes in order to process sequences.[20] 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.

6.6.1 Performance Model

Although the techniques of earlier sections produce *correct* incremental parsers for any grammar accepted by the parser generator, the choice among grammars accepting the same language matters greatly for incremental *performance*. We now examine the assumptions that accompanied the performance analysis of the algorithms in Sections 6.3 and 6.4.

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

[19]Note that the most powerful transformations—those involving right-recursive expansions of sequences [44]—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 symbol after the final sequence element 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 [85] cannot be used; these algorithms treat sequences in an inherently batch fashion.

[20]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 repetitive sections immediately before processing a commit (Chapter 3). Tools should perform on-line re-balancing only when performance would be severely degraded by waiting until the completion of the edit.

$$s \rightarrow L^*$$

10a. Grammar with sequence notation.

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

10b. Conventional left-recursive formulation.

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

10c. Non-deterministic expansion of sequences.

10d. Right-recursive tree structure.

10e. Balanced tree structure.

Figure 6.10: 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 6.11.

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 $(A|B)X^+$:

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

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

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

Figure 6.11: 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 non-empty, comma-separated list of identifiers, e.g., would be written as `idlist→ID+[',']`.

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.[21]

6.7 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 (Section 4.5). Figure 6.1 indicates the set of nodes that can be retained through explicit reuse calculation in the common case of changing an identifier spelling.

6.7.1 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 6.7.3.

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 referred to as *bottom-up* reuse.[22]

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 6.4.2). 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 6.12 illustrates both types of reuse paths.

Several incremental parsing algorithms have tried to capture a subset of top-down reuse by implementing a *matching condition* [58, 61, 76]. 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 [36]) 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.[23]

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.

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

[22] Recall from Section 3.2 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.

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

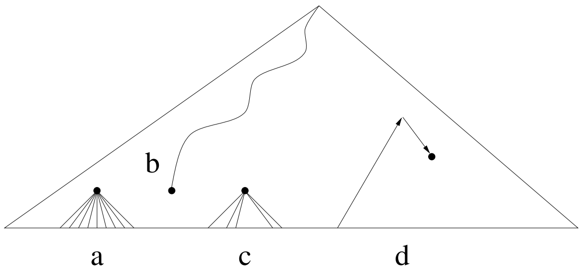


Figure 6.12: Illustration of reuse paths. In each case, the circle represents a reused node, and the lines indicate the reuse path the justify its retention in the current version by linking it to some base case.
(a) Unambiguous bottom-up reuse (reused contents). (b) Top-down reuse (reused context). (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.

6.7.2 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.[24] Ambiguous bottom-up reuse can also create new starting points from which top-down reuse paths can originate (Figure 6.12d).

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.)

6.7.3 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 6.13 illustrates this test.

Ambiguous bottom-up reuse can be computed in a similar manner by relaxing the reuse condition (Figure 6.14). 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 6.15 illustrates this process. *Reachability analysis* discovers nodes in the previous version of the tree that have been eliminated in the new version (Section 3.2); 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 6.15 are required to support the ambiguous reuse model.

[24]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;
}
```

Figure 6.13: 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 3.2.

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);
}
```

Figure 6.14: Computing ambiguous bottom-up node reuse at reduction time. This method differs from that of Figure 6.13 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 () {
  process_deletions(UltraRoot);  Section 3.2.
  top_down_reuse_traversal(root);
}
```

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);
  }
}
```

Figure 6.15: 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. `process_deletions` 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.

6.7.4 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 6.6, 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 6.1, this simple method results in only one changed node in the entire tree: the modified token.[25]

6.8 Conclusion

This chapter 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.

[25] 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.

Chapter 7

Non-deterministic Parsing and Multiple Representations

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

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

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

7.1 Introduction

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

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

Rather than lament the design of these languages, we address the underlying issue by extending the language model, producing a framework that allows existing formalisms to apply to a wider variety of languages. Our solution utilizes a new intermediate representation (IR) for the early portions of the (possibly incremental) compilation pipeline: the *abstract parse dag* allows multiple interpretations to be represented directly and efficiently. The familiar pass-oriented compiler organization is supported, even in incremental settings, by allowing ambiguities to be resolved at different stages of the

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

Figure 7.1: A simple example of ambiguity in C and C++. In this case, type information is necessary for disambiguation: the middle two lines can be either declarations or function calls, depending on how a and c have been declared previously in enclosing scopes.

analysis. Semantic filters address the 'feedback' problem (syntactic structure dependent upon semantic information) arising in C and Fortran. Parsing filters [53] address such problems as the declaration/expression ambiguity in C++ [28] and the 'off-side' rule in Haskell [46]. We describe mechanisms for applying both types of resolution using existing formal techniques, such as attribute grammars, while also permitting *ad hoc* resolution. Pre-compiled filters such as precedence and associativity declarations in `bison` [4] are supported in a uniform fashion. In the presence of missing or malformed program text, multiple interpretations may be retained indefinitely as a direct expression of the possibilities.

We have developed a novel algorithm for incremental, non-deterministic parsing to (re)construct this IR. The parser accepts all context-free grammars: generalized LR parsing [79, 92] is used to support non-determinism and ambiguity, eliminating restrictions on the parsing grammar and the attendant need for abstraction services. Shifting of entire subtrees via state-matching [49] provides efficient incremental behavior, and explicit node retention minimizes the work of subsequent analysis passes. (Together they also ensure the preservation of user context and program annotations.) Lookahead information is dynamically tracked and encoded in parsing states stored in the nodes, eliminating the space overhead of previous approaches that require persistent maintenance of the entire graph-structured parse stack [30].

As an example of an inherent context-free syntax ambiguity addressed by this representation, consider the syntax of C. Figure 7.1 illustrates a case where the interpretation of several lines is context-sensitive, i.e., 'static semantic' analysis is needed to resolve the ambiguity.[1] A similar problem arises in C++, Fortran, Oberon, and other languages. This problem arises whenever the natural context-free syntax depends on non-local type information [103].

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

Encoding alternatives for later resolution is useful in a number of stages in the compilation pipeline. Lexical decisions are often deferred until parsing or semantic analysis by having the lexer recognize only equivalence classes of tokens. Visser [97] makes this integration explicit for a batch system by using a single GLR parser for both lexical *and* context-free analysis. This approach can be made incremental using the techniques we describe. Code generation also benefits from retaining multiple representations until additional information has been gathered. Giegerich [37] applies context-sharing in this domain to intersperse code selection and register allocation.

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

The remainder of this chapter is organized as follows. In Section 7.2 we describe the basic form of the program representation, concentrating on the handling of alternative interpretations. Section 7.2 also summarizes empirical studies demonstrating the highly localized nature of ambiguity in programs and the minimal space overhead achievable through sharing. In Section 7.3 we consider in detail the construction of our program representation using an incremental, non-deterministic parser. We introduce a performance model and analyze the asymptotic behavior of the parser to demonstrate the efficiency of incremental updates. We conclude this section with a return to the issue of sharing in the abstract parse dag, demonstrating optimality and correctness properties unique to our method. A trace of the parser actions on a small C++ example is given

[1] Batch systems typically handle this problem by having the lexer query the symbol table in order to separate identifiers into two distinct categories. Attribute-influenced parsing [51, 84] is a combination of LR parsing and a restricted class of attribute grammars that addresses the same problem in a formal way. Neither of these solutions can be applied to an incremental setting where non-trivial subtrees appear in the parser's input stream.

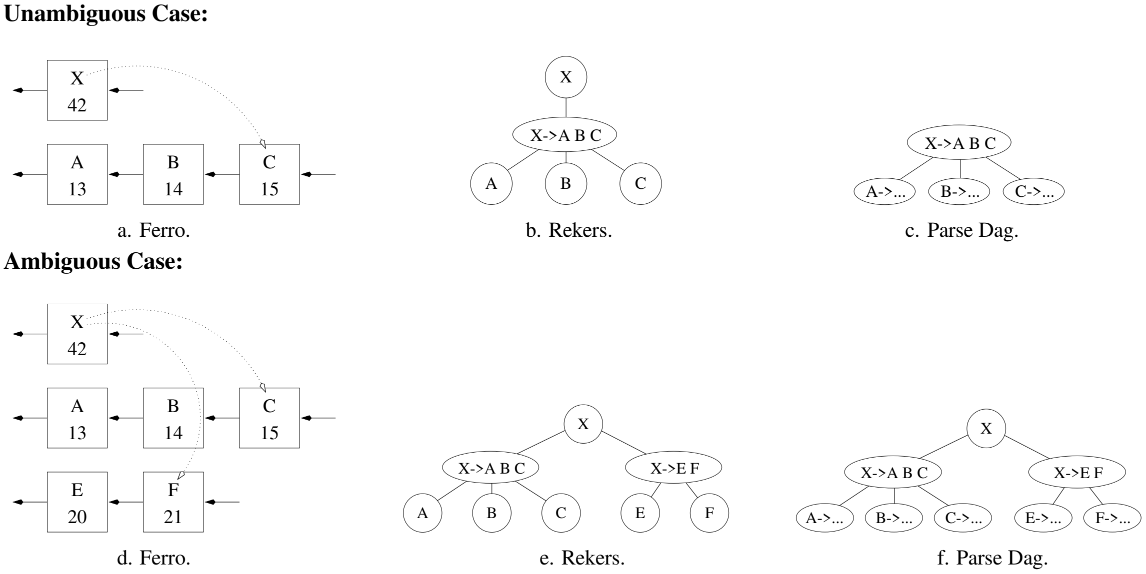


Figure 7.2: Comparison of the abstract parse dag to other proposed representations. The grammar productions illustrated are X→ABC | EF. Ferro and Dion's approach (a) makes the GSS itself persistent; this requires semantic attributes associated with a production (right-hand side) to be attached to a constellation of nodes rather than an individual object. Rekers' representation (b) is more like a classic parse tree but separates the symbol (phylum, left-hand side) and rule (production, right-hand side) into separate nodes. This imposes significant overhead, since the vast majority of the program is deterministic. Our approach represents the deterministic portions of the tree in the conventional manner (c), using Rekers-style splitting only where multiple representations actually exist (f). (Not shown are the additional state collections required by the Ferro and Dion approach or the problems with under- and over-sharing of epsilon productions eliminated in the abstract parse dag.)

in Section 7.4. Mechanisms for disambiguation at various points in the analysis phase—particularly semantic disambiguation involving type information—are presented in Section 7.5. Implementation details and empirical comparisons between deterministic parsing/parse trees and non-deterministic parsing/abstract parse dags are given in Section 7.6. A discussion of future work and our conclusions end the chapter. The incremental GLR parsing algorithm is provided in Appendix B.

7.2 Representing Ambiguity

A phase-oriented incremental system can succeed only if the intermediate representation explicitly represents unresolved ambiguities. The *abstract parse dag* is similar to a parse tree except that a region may have multiple interpretations. This section describes the representation itself; subsequent sections describe its construction, via non-deterministic parsing, and the resolution of ambiguities expressed through this IR.

In the presence of ambiguity, many parse trees potentially represent the program. To avoid exponential blowup, this entire forest is collapsed into a single, compact data structure. *Subtree sharing* merges isomorphic regions from different trees, and requires no special changes—each instance of a production is represented by a single node, just as in a parse tree. Merging *contexts*,[2] however, requires a new type of node to indicate the choices. A *symbol node* represents a phylum (left-hand side) instead of an entire production; its children represent the possible interpretations of their common yield. In the case of a correct program, later stages of analysis will disambiguate the program by selecting exactly one child of each symbol node. Figure 7.2 illustrates the distinction between symbol and production nodes and compares our representation to other proposals. Figure 7.3 shows the abstract parse dag corresponding to the example in the introduction.

If the number of alternate interpretations at a single point is large, the children of a symbol node can be represented as a balanced binary tree to ensure the performance characteristics described in Section 7.3.3. In practice, however, the number of alternatives is effectively bounded and a simple list provides sufficiently fast access.

In a typical batch compiler, a grammar from a restricted grammar class is used to produce a parser for the concrete syntax. A separate (often implicit) grammar defines the abstract syntax representation of the parsed program after artifacts of the concrete parse have been removed. GLR parsing enables a *single* grammar to formally define both the representation

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

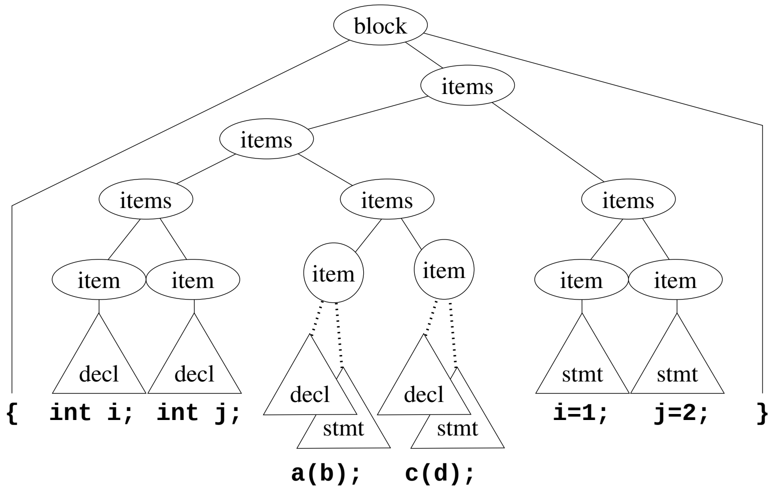


Figure 7.3: Representation of ambiguous structure in the abstract parse dag. This is the result of parsing the example in Figure 7.1 as a C++ program. Most nodes represent both productions and symbols. *Choice points*, shown as circles, represent only symbols; their children comprise the alternative interpretations. In this case the shared subtrees are trivial—they are the terminal symbols in the ambiguous region. The structure shown represents a simplification of the complete grammar.

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

Table 7.1: Programs used in this study. The first eight are from SPEC95. `idl` is the SunSoft IDL front end and `ensemble` is our prototype ISDE.

and the mechanism that builds it: support for multiple syntactic interpretations and non-deterministic parsing permit arbitrary CFGs to be used in describing the language. This generality allows the grammar to serve as a pure definition of the resulting structure, rather than requiring it to conform to the restrictions of some particular parsing class.[3] Since our parse dag representation inherits this benefit of GLR parsing, we refer to it as 'abstract'. (We will sometimes omit this modifier.)

The abstract parse dag differs from the ordinary shared forest discovered by a GLR parser: Instances of productions are always represented by individual nodes, and sharing of both subtrees and contexts is optimal. We return to issues of sharing in Section 7.3.4 after explaining incremental GLR parsing.

7.2.1 Space Overhead for Ambiguity

Cognitive studies suggest that localization of ambiguity is an inherent property of natural languages, a constraint imposed by limitations on short-term memory [62, 69]. Our studies find an identical result for programming languages.[4] Since an abstract parse dag exploits localization of ambiguity through the sharing of subtrees and contexts, the increase in space

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

[4] This property was indirectly measured by Tomita [92] and Rekers [79], who compared the speed of a batch GLR parser to Earley's algorithm [27] on natural and programming language grammars, respectively. Both authors concluded that grammars are 'close' to LR(1) in practice, and therefore GLR parsing exhibits linear behavior despite its exponential worst-case asymptotic result.

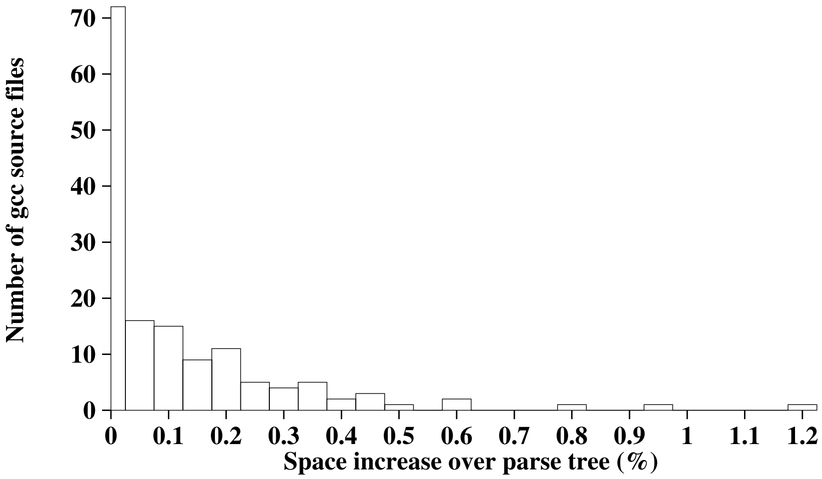


Figure 7.4: Distribution of ambiguities by source file in gcc. This histogram groups the source files of gcc according to the amount of syntactic ambiguity they possess. The syntax of C++ was used to determine these counts; the percentages would be lower using a C grammar, due to the more restrictive statement syntax of that language. All ambiguities are semantically resolved (the 'typedef problem'). They consist of two interpretations each, and share only terminal symbols.

required relative to a fully disambiguated parse tree provides an ideal measure of the amount of ambiguity (as well as the space overhead of adopting our IR). For the suite of C and C++ programs in Table 7.1, we measured the increased space consumption required to represent the multiple interpretations of each syntactically ambiguous construct. The increase is relative to the parse tree produced by a batch compiler (using semantic feedback to the lexer and with the corresponding ambiguity in the grammar resolved through different identifier namespaces). The average increase for each program in the suite is shown in the final column of Table 7.1. Figure 7.4 shows the ambiguity distribution by source file for gcc.

7.3 Constructing the Abstract Parse Dag

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

7.3.1 Generalized LR Parsing

Batch GLR parsing [73, 79, 92] is a technique for parsing arbitrary context-free grammars that utilizes conventional LR table construction methods. Unlike deterministic parsers, however, a GLR parser permits these tables to contain conflicts: when a state transition is multiply defined, the GLR parser simply forks multiple parsers to follow each possibility. In the case of a deterministic parse requiring additional lookahead, all but one of these parsers will eventually terminate by encountering a syntax error. In the case of true ambiguity, multiple valid representations will be discovered. In both cases, the *graph-structured parse stack* (GSS) represents the combined parse stacks compactly. This sharing is made possible by having the GLR parse proceed breadth-first: each terminal symbol is shifted simultaneously by all active parsers in the collection. Figure 7.5 illustrates a GLR parser processing the non-LR(1) grammar of Figure 7.6.

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

7.3.2 Incremental GLR Parsing

We now turn to the construction of an *incremental* GLR (IGLR) parser that can parse an arbitrary CFG non-deterministically, while simultaneously accepting non-trivial subtrees in its input stream. The abstract parse dag is (re)created during parsing; Section 7.3.4 explores this process in more detail. Section 7.4 contains a sample trace of the IGLR parser's actions using our running example and a simplified C++ grammar.

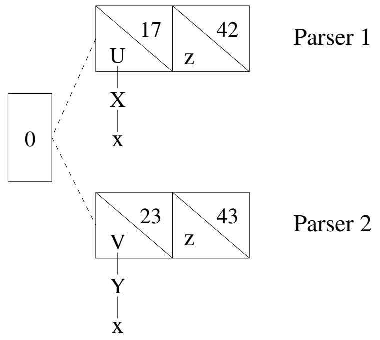


Figure 7.5: Illustration of non-determinism in a GLR parser. When the grammar is ambiguous or requires lookahead greater than that of the table construction method (typically a single terminal), a GLR parser will *split* into two or more parsers. Here two parsers are being used in a region requiring two terminals of lookahead with an LR(1) table. (The grammar appears in Figure 7.6.) In this case the parse is non-deterministic but unambiguous: when sufficient lookahead has been scanned dynamically, the GLR automaton will collapse back to a single parser. In cases of true ambiguity, multiple interpretations are preserved in the resulting abstract parse dag.

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

As in previous GLR algorithms, we employ a graph-structured parse stack (GSS) to permit non-deterministic parsing. During parsing, deterministic behavior is assumed to be the common case. (Sections 7.2.1 and 7.6 validate this assumption through empirical measurements.) As with a deterministic state-matching parser, each node of the parse dag requires an additional word of storage to record the parse state in which it was constructed. LALR(1) tables are used to drive the parser: not only are they significantly smaller than LR(1) tables, but they also yield faster parsing speeds in non-deterministic regions [57] and improved incremental reuse in deterministic regions (due to the merging of states with like cores).[5]

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

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

Additional (dynamic) lookahead is required only when several parsers are simultaneously active. The IGLR parsing algorithm tracks this condition with a boolean flag. After shifting the lookahead, the flag is set to true if there are multiple active parsers. The flag is also set to true when a parse table interrogation returns multiple actions. During a reduction, the state value recorded in the newly created dag node is the state of the single active parser, if the flag is false, and the value representing all non-deterministic states (and thus the use of additional lookahead), if the flag is true. Figure 7.6 shows a simple case where dynamic lookahead is used by our IGLR parser to analyze an LR(2) grammar using LR(1) tables.

When both the previous state (preserved in the root node of the lookahead subtree) and the current state are deterministic, parsing proceeds as in Chapter 6. Shifted subtrees may contain non-deterministic areas as long as they are not exposed. Subtrees containing modifications (textual and/or structural edits) are decomposed to expose each change site. Subtrees from non-deterministic regions are similarly broken down, triggered by a failure of the normal state matching test. If a conflict is encountered, the parser splits just as in batch GLR parsing, and subtrees in the input stream are fully decomposed until a deterministic state is re-established (see the `shifter` routine in Appendix B).

Shifting an unmodified, non-trivial subtree condenses a sequence of transitions by the corresponding batch GLR parser. The portion of the abstract parse dag reused when the incremental algorithm shifts a non-trivial subtree reflect any splitting or merging that would occur in the GSS of the batch algorithm as it parsed the subtree's terminal yield. The correctness of

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

```
A → Bc | De
B → Uz
D → Vz
U → X
V → Y
X → x
Y → x
```

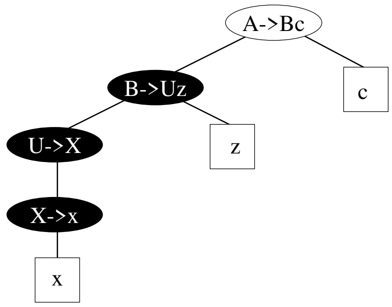


Figure 7.6: Tracking lookahead information dynamically. This example illustrates a grammar that requires two tokens of lookahead. A GLR parser based on a single-lookahead table will require non-determinism to parse the sentence `xzc`. Since the grammar is unambiguous, a unique parse tree results after `c` is read. The unsuccessful parser is discarded. Black ellipses indicate nodes for which increased lookahead must be recorded during parsing; note that they coincide with reductions performed while more than one parser was active. Nonterminals representing reductions in a deterministic state (`A→Bc`) require only the implicit (one token) lookahead; they are marked with the (singleton) parser's state when they are shifted onto its parse stack.

skipping the intermediate steps is guaranteed in deterministic states by the usual incremental context checks, and in non-deterministic states (which are treated as an equivalence class) by the restriction to terminal lookaheads. The correctness of incremental GLR parsing can then be established by an induction over the input stream.

Our approach differs significantly from the non-deterministic PDA simulator of Ferro and Dion [30], which uses the GSS itself as the persistent representation of the program. Their representation requires more space than our parse dag, in part because unsuccessful parses (used to overcome lookahead limitations) must be retained for the sake of future state comparisons. (In Figure 7.5, the portion of the GSS constructed by Parser 2 must be kept, even though it represents an unsuccessful search.) Their algorithm also makes state comparisons and semantic attribution more expensive, since both must refer to a *collection* of nodes.

As with deterministic parsing, IGLR parsing can be extended to retain existing program structure through *node reuse* [58, 76, 99]. Both ambiguous and unambiguous reuse models are valid for abstract parse dags, and both bottom-up and top-down reuse mechanisms can be applied. (For on-the-fly bottom-up reuse, we advocate retaining a single, shared list of reused nodes; maintaining separate lists when multiple parsers are active imposes a performance and complexity cost for minimal gain in the number of reused nodes.)

7.3.3 Asymptotic Analysis

The IGLR parsing algorithm works for any context-free grammar and, like GLR parsing, is exponential in the worst-case [50] but linear on actual programming language grammars. To ensure incremental performance that improves on batch parsing, we impose the same restrictions on the grammar and the representation of associative sequences in the abstract parse dag as in deterministic parsing (Section 6.6).

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

Given this assumption regarding the form of the grammar and the representation of the abstract parse dag, we can analyze the time performance of the IGLR parser. In the typical case where the left and right context of a subtree are unchanged, a state-matching algorithm will shift that subtree in $O(1)$ time. In the event the context *has* changed, a valid subtree containing M nodes can be shifted in $O(\lg M)$ steps by reconstructing its leading or trailing edge. Reductions and the deterministic right context check are often accomplished in $O(1)$ time using the following subtree; in the worst case the following terminal symbol is located in $O(\lg M)$ steps. Locally non-deterministic regions are reconstructed in their entirety, but our assumption that the size of such regions is effectively bounded (Section 7.2.1) implies a constant bound on the time to parse them. The result is a typical parsing time of $O(t + s \lg N)$, for t new terminal symbols and s modification sites in a tree with N nodes, and $O(t + s(\lg N)^2)$ time in the worst case. (Empirical results are discussed in Section 7.6.)

7.3.4 Correct and Optimal Sharing

Our approach treats the GSS as a transient data structure of the parser, using it to construct the abstract parse dag in the same way deterministic parsers construct a concrete parse tree with the help of a parse stack. However, the connection is more complex than in the deterministic case. In this section we discuss the removal of parsing artifacts from the shared parse forest discovered by GLR methods to produce the representation described in Section 7.2.

The parse forest produced by GLR parsing results in both over- and under-sharing, complicating (in some cases precluding) the application of existing methods for semantic attribution and similar tools. GLR parsing as originally defined [92] results in *under*-sharing in the shared parse forest when isomorphic subtrees with the same yield are created in different states (i.e., by different parsers) due to left or right contextual restrictions.[6] Rekers corrects under-sharing in his batch GLR parser by merging nodes that have identical yields [79]. Merging is performed separately for both symbol and 'rule' (production) nodes. The same approach can be applied in our algorithm, since non-deterministic regions are reconstructed atomically.

A different problem exhibited by GLR algorithms is *over*-sharing. A GLR parser does not distinguish non-determinism to acquire additional lookahead information from its use in parsing ambiguous phrases. In most cases, non-determinism for dynamic lookahead results in deterministic (and unshared) structure in the parse tree, since unsuccessful parses eventually terminate. In the GSS, however, sharing needed to handle certain types of grammars with ϵ-productions results in sharing in the parse tree *even for unambiguous grammars* [73]. We consider this a flaw; among other problems, it prohibits semantic attributes or annotations from being uniquely assigned to productions with a null yield, since separate instances may not exist in the parse tree. (Rekers' algorithm exacerbates this problem by merging additional null-yield subtrees, violating left-to-right ordering.) We correct this problem by adding a post-pass that incrementally duplicates any null-yield subtrees updated by the parser. Since a unique maximal sharing of these subtrees does not necessarily exist, this is the only approach that is consistent, correct, and practical. Node reuse strategies (Section 6.7) can be used to prevent unnecessary recreation of these and other subtrees.

7.4 Sample C++ Trace

In the example shown in Figure 7.7, we trace the parser's actions in constructing the dual interpretations of the 'typedef' problem in C++, using a simplified grammar. Consider the input stream as it appears in (1), and suppose the semicolon has been deleted and then re-inserted. The region to the left of the semicolon was an ambiguous `item`; the edit to the semicolon causes the parser to discard the non-deterministic structure and read `id(id)` as terminal symbols.

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

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

In (13) context sharing occurs as the two parsers merge into a single parser. The `item` node shown on top of the stack is a symbol node;[7] its two children represent the two interpretations of its terminal yield. Now that the state is once again deterministic, the parser returns to shifting entire subtrees.

7.5 Resolving Ambiguity

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

---

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

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

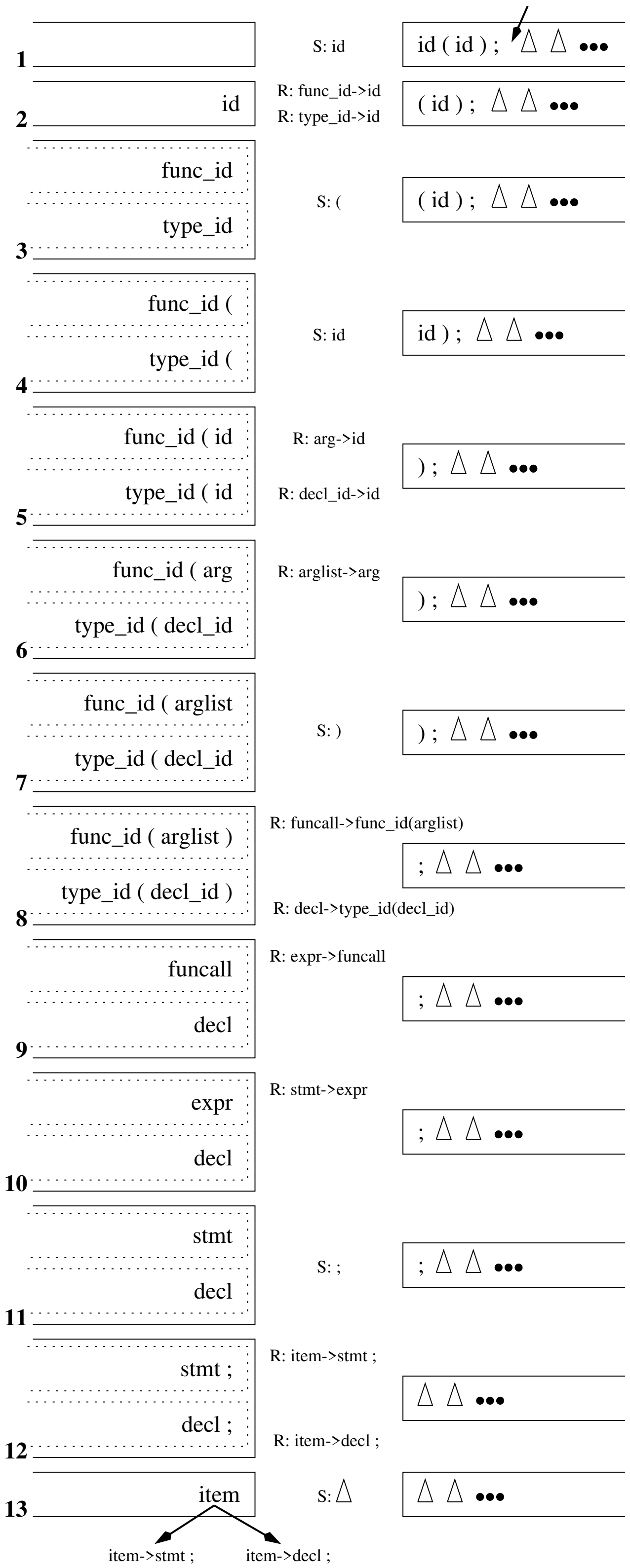


Figure 7.7: Sample trace of IGLR parser on a small example.

7.5.1 Syntactic Disambiguation

Static syntactic filters, in conjunction with ambiguous grammars, are used frequently in compiler construction. Examples include the operator precedence and associativity specifications in `bison` [4] as well as techniques associated with a particular parse table construction algorithm, such as 'prefer shifting'. Such methods can be applied at language specification time by selectively removing conflicts from the parse table, and therefore do not result in non-deterministic parsing or multiple representations. Since state-matching incrementalizes transitions in the pushdown automaton, any disambiguation statically encoded in the parse table is supported by the IGLR parser.

When the selection of a preferred interpretation cannot be determined *a priori* based on the left context and the implicit ('builtin') lookahead, a *dynamic* filter is required. For example, the syntactic ambiguity in C++ expressed as 'prefer a declaration to an expression' requires a dynamic filter, since competing reductions cannot be delayed until sufficient lookahead has been accumulated [28]. The abstract parse dag allows ambiguities of this form to be encoded using multiple interpretations; an incremental post-pass can then select the preferred structure by directly applying rules such as the one above.[8] Syntactic disambiguation of this form can also take place on the fly, provided it occurs only in a deterministic state to avoid contaminating the dynamic lookahead computation. Unlike Ferro and Dion [30], we do *not* retain interpretations eliminated by syntactic filters.

In general, disambiguation specifications [42, 53] can be compiled into a combination of static and dynamic filters. Encoding as much filtering as possible at language specification time decreases both the size of the representation and the analysis time. (This contrasts with existing batch GLR environments, which perform *all* syntactic filtering dynamically [79, 92], and thus require quadratic space for each expression, in contrast to the negligible increases we report in Section 7.2.1.)

7.5.2 Semantic Disambiguation

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

Figure 7.8 illustrates the sequence of events. After context-free analysis is complete, the first stage of semantic analysis is applied to process `typedef` declarations. Type names introduced by such declarations are gathered into a *binding contour*, which is then propagated throughout the scope. (This information will be inherited by both children of a symbol node, reaching each identifier in an ambiguous region twice.) In a correct program, the binding contour's contents uniquely determine the namespace for each identifier.

With identifier namespaces decided, disambiguation *per se* can take place: 'parsing' is completed by propagating the namespace decision throughout the ambiguous region. Boolean semantic attributes indicate nodes filtered out of the parse dag in the unwanted interpretation. Since all syntactic and semantic ambiguities have now been resolved, each symbol node can be logically identified with its single remaining child in subsequent passes, allowing tools to treat the result as a normal parse tree.[9]

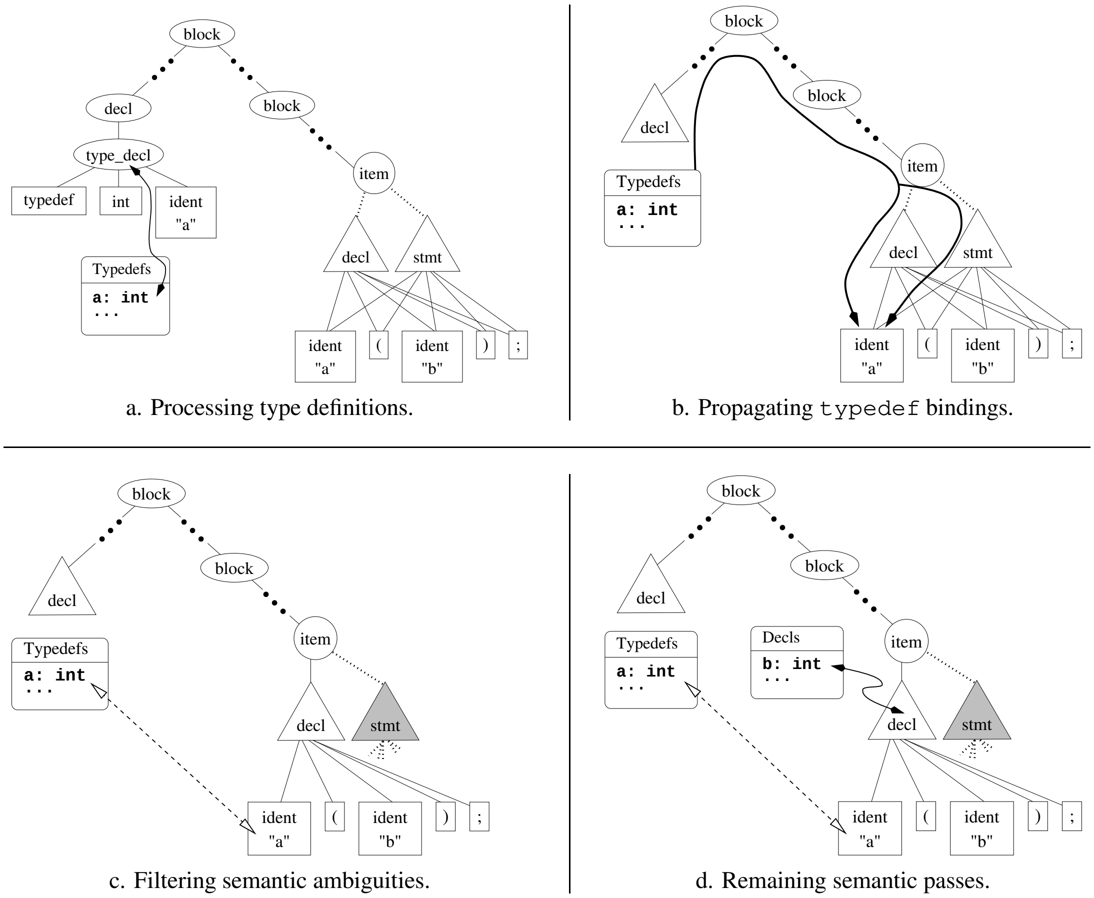
The order of the passes is the same for both batch and incremental scenarios. In the incremental case, each stage inspects or updates only those portions of the program that have changed or could possibly be affected by preceding changes [63]. An interesting case occurs when a `typedef` declaration is removed: Binding information stored in semantic attributes allows the former uses of the declaration to be efficiently located. At each use site, the interpretation of the ambiguous region will change from a variable declaration to a function call as the namespace of the region's initial identifier is altered. Note that the use sites themselves require no action from the parser; other attributes of the reinterpreted regions are re-evaluated as semantic analysis progresses.

7.5.3 Program Errors

When the program is correct with respect to the language description (and the language as a whole is unambiguous) a single structural representation will eventually be discovered. In the presence of semantic errors, such as missing, malformed, or inconsistent declarations, it may not be possible to determine a single interpretation of the entire structure. In such cases the abstract parse dag maintains multiple interpretations persistently; future edit/analysis cycles may eventually correct the

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

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



a. Processing type definitions.

b. Propagating `typedef` bindings.

c. Filtering semantic ambiguities.

d. Remaining semantic passes.

Figure 7.8: Illustration of semantic disambiguation. This shows our running example (using C++, although the situation is similar in both C and Fortran) during the semantic analysis passes. In (a) the basic context-free analysis has been completed, and the first stage of semantic analysis now resolves `typedef` definitions. In (b) this binding information is propagated to the ambiguous regions, allowing the selection of the appropriate namespace for each identifier. In (c) disambiguation *per se* occurs, as the unwanted interpretation is filtered out (it is retained in case future edits reverse the decision). In (d) semantic analysis continues, using the embedded tree discovered by stages a–c. (Note: The right-hand side of production labels are omitted.)

errors and allow the resolution to succeed. These regions are re-evaluated by the parser only when they are modified and by semantic analysis only when they require re-interpretation.

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

Errors in the context-free syntax may also occur and are detected in the usual fashion: when no parser can successfully shift the (terminal) lookahead symbol. History-sensitive error recovery for deterministic parsing is covered in Chapter 8; the only change required to support IGLR parsing is an extension of the isolation boundary test to ensure that each non-deterministic region is treated as an atomic unit: partial update incorporation within such a region is not permitted. (This has no practical effect on the efficacy of the recovery, due to the small size of these regions in actual programs.)

7.6 Implementation and Empirical Performance

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

Despite the slightly less efficient stack representation used for GLR parsing relative to deterministic parsing, the IGLR parser performs an initial ('batch') parse nearly as fast as its deterministic counterpart. C,[10] Java, and Modula-2 programs were parsed with both parsers, and yielded an average of 12% overhead due to parsing *per se* for the deterministic parser, compared with 15% for the IGLR parser. Most of the remaining time was spent in constructing the nodes. In incremental tests (self-cancelling modifications to individual tokens, parsing after each such change) the difference in running times for the two parsers was undetectable.

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

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

7.7 Extensions and Future Work

Techniques for expressing both syntactic and semantic filtering in a uniform language would both simplify the language description process and allow optimized performance by applying resolutions at the earliest possible stage. Visser uses priorities and tree patterns to produce static filters [95], but further work is needed.

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

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

7.8 Conclusion

This chapter provides a mechanism for applying the open, pass-oriented framework of batch analysis tools to incremental environments. A new IR, the abstract parse dag, is introduced to model ambiguity in programming language analysis. Circular analysis dependencies as they exist in C, C++, Fortran, and other common languages are eliminated by the ability to apply disambiguation filters at any point in the analysis process. Arbitrary CFGs may be used to describe the form of the parse dag, as well as to produce fast incremental parsers based on our IGLR algorithm. Optimal and correct subtree and

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

context sharing in the abstract parse dag are obtained by removing parsing artifacts from the shared parse forest. Empirical measurements demonstrate the space efficiency of our representation and the time efficiency of our reconstruction methods, both of which exploit an underlying language property: localized non-determinism.

Chapter 8

History-Sensitive Error Recovery

In this chapter we present a novel approach to incremental recovery from lexical and syntactic errors in an ISDE. Unlike existing techniques, we utilize the history of changes to the program to discover the natural correlation between user modifications and errors detected during incremental lexical and syntactic analysis. Our technique is *non-correcting*—transformations intended to restore consistency between the structure and text of the program will not incorporate invalid modifications, while still permitting valid modifications to be applied. Errors are presented to the user simply by highlighting his invalid changes.

The approach is automated—no user action is required to detect or recover from errors. Multiple textual and structural edits, arbitrary timing of incremental analysis, multiple errors per analysis, and nested errors are supported. History-based error recovery is language independent and is compatible with the methods for incremental lexing and parsing described in the preceding chapters, adding neither time nor space overhead to those algorithms. Effective integration with the environment's history services ensures that other tools can efficiently discover regions of the program (un)affected by errors, and that any transformations of the program required to isolate or present errors are themselves efficiently reversible operations.

8.1 Introduction

Syntactic error recovery in batch systems is essentially a solved problem, involving a heuristic computation based on the configuration of the parser when the error is detected [22]. The best methods known rely on the ability to delay actions or reproduce part of the parse on demand, so that a variety of repairs may be tried at locations other than the detection point [14, 19, 38]. Since the recovery routine has no knowledge of the user's changes with respect to previous versions of the program, it attempts to correlate the problem with the detection point by comparing the results of different repairs. When the error is significantly complex or distant from its detection point, a less informative 'second stage' recovery may be needed.

In an ISDE, errors can arise as they do in batch systems, since arbitrary modifications to the text and structure of the program are permitted. Errors introduced by changes will be discovered when the user next requests incremental analysis. Many types of problems can occur, including a variety of static semantic errors (type inconsistencies, missing definitions). However, errors associated with the lexical and context-free syntax play a special role in an ISDE: the structural representation is a fundamental data structure, and language specifications typically do not prescribe the representation of erroneous programs.[1] Thus the system is faced not only with the task of effectively detecting and reporting any errors, as in a batch compiler, but also with integrating some representation of the problem into the persistent, structural representation of the program. No satisfactory approach to this problem has previously been available.

Several systems have tried to minimize or circumvent the difficulty of error handing in an ISDE by limiting the class of modifications available to the programmer [11]; in the extreme case, only structural operations that preserve all correctness properties are permitted [72]. Our approach is the other extreme: we place *no* restrictions on the editing model, allowing arbitrary textual and structural modifications and arbitrary timing of the analysis. Multiple errors, including nested errors, may exist simultaneously and do not preclude the incorporation of other modifications. Errors may persist indefinitely; the environment must tolerate the presence of any invalid or inconsistent material and continue to provide as much functionality as possible [93]. The goal of the environment is to isolate problematic regions, inform the user of their location, and provide assistance by explaining the reason these modifications could not be adopted successfully.

Our approach is fully automatic—no user intervention is required to detect errors, and the user is free to correct errors in any order, at any time. In contrast, several researchers have addressed error recovery in an ISDE by attempting to utilize

[1] Static semantic errors, on the other hand, can be represented without leaving the framework of the attribute grammar or similar formalism.

Initial (correct) program.

```
int f () {
  g(a + b);
  if (c == 3) c = 4;
  else c = 5;
}
```

Introducing three errors.

```
int f () { {  Inserted extra opening brace
  g(a + b );   Deletion
  if (c == 3) c = 4;   Deletion
  else c = 5;
}
```

Result: only one error is detected.

```
int f () { {
  g(a + b  ERROR
  else c = 5;
}
```

Figure 8.1: An example where batch non-correcting techniques fail. The first error (extra opening brace) is hidden by subsequent problems. The second error, a deletion, is detected, but the non-correcting nature of the recovery precludes discovery of the third error (`else` without matching `if`), since what remains after the deletion is a valid substring. Our approach discovers all three errors *without* attempting to correct the program; the visual presentation would be similar to the second version above with the explanatory text removed.

the interactive nature of the environment [6, 49, 87]. Unfortunately, these approaches all demand direct user intervention at each error site and therefore impose an unnecessary serialization on the analysis and the user's (manual) recovery actions. A few research and commercial environments support unattended incremental error recovery in the context of incremental parsing [8, 48, 83] but there has been little discussion or analysis of the technologies employed. *No* existing systems make use of the vast amount of information available in the development log being maintained by the environment.

The central idea in our approach is to recognize that some user modifications introduce (locally) valid changes while others do not: modifications successfully incorporated into the structure and content of the program representation are retained, while invalid changes remain in their 'unanalyzed' form. This is a *non-correcting* strategy: unlike most automated recovery schemes, it does not attempt to guess the programmer's intention. The well-known drawbacks of correcting strategies are avoided: no conjectures are necessary, spurious repairs never arise, and no heuristic 'language tuning' is needed. The correctness of *any* repair we perform can be established easily, even in a multilingual, incremental setting.

Non-correcting approaches [20, 80, 82] have not received much attention in batch compilers. Despite the theoretical advantages described above, the practical limitations imposed by a batch setting cause even the best of these approaches to be less useful than correcting methods. The most critical shortcomings involve errors in bracketing syntax and the fact that the initial error typically obscures detection of subsequent problems, since the following text is often a valid substring in *some* sentence.[2] Figure 8.1 illustrates these deficiences.

Our recovery scheme overcomes these deficiences by using *historical information*: the sequence by which the programmer arrived at the current state affects the treatment and reporting of errors [104]. Changes recorded in the development log permit comparisons between the current and previous versions of the program, providing a guide for determining the source of a given problem *in terms of the user's own modifications*. This approach can discover the relationship between the point where an incorrect change was applied and the point where the error was finally detected even when they are far apart or separated by intervening errors.

Determining the relationship between changes to the program and subsequent errors is a novel and powerful tool for error handling. While the best known batch correcting recoveries handle the example in Figure 8.1 better than their non-correcting counterparts, they will typically fix the initial error by adding an extra closing brace at the end of the program. Our approach instead 'corrects' the problem by refusing to insert the extra opening brace into the structural representation of the program (although it remains visible in the text)—a better and more comprehensible response given the actual change made by the user.

[2] Right-to-left substring parsing (*interval analysis* [9, 82]) has been proposed to further constrain the location of detected errors, but common mistakes can result in intervals so large that the user must still locate the problem manually. Interval analysis does not address the detection problems of batch non-correcting recoveries.

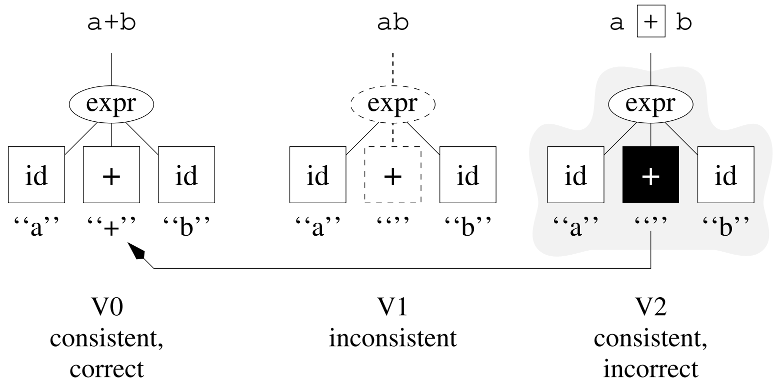


Figure 8.2: The recovery and presentation of a simple error. In V1, the dashed lines indicate the path from the root to the site of the unincorporated modification. In V2, the isolated region is indicated by light shading; the token shown in black has been marked to indicate that it contains an invalid textual deletion. The visual presentation corresponding to each version is shown above the subtree. The arrow from V2 to V0 indicates the structural correlation that allows the presentation system to display the deleted text responsible for the problem.

The dependencies between program components induced by lexical and syntactic analysis methods provide a natural way to discover and limit the scope of a given error. This *isolation* process makes it possible to treat unrelated errors independently and allows correct modifications outside the isolated regions to be successfully incorporated. Isolation is computed incrementally, by comparing the current (partial) structure of the program to the previous structure stored in the development log.

Not all of the modifications within an isolated region are necessarily incorrect, and even a well-chosen isolation region can be very large. Thus some mechanism to detect legal modifications within an isolated region is needed. Two techniques are used: *retention of partially analyzed regions*, which avoids discarding legal updates prior to the detection point, and *out-of-context analysis*, which applies incremental analysis techniques to modified subtrees within the isolated region. Together, these techniques typically allow legal modifications to be incorporated, even in close proximity to one or more errors.

History-sensitive error recovery is language independent, using only information derived from the grammar and the user's own editing actions. The recovery is guided by existing mechanisms for lexical and syntactic analysis; the language designer is not required to provide additional specifications in order for error recovery and reporting to function.[3] The approach is compatible with incremental lexing (Chapter 5) and both deterministic (Chapter 6) and non-deterministic (Chapter 7) parsing.

Errors are represented in a simple fashion: the invalid modifications are simply maintained as unincorporated edits. This suggests a presentation of errors that is at once trivial and powerful: the unincorporated edits are visually distinguished to indicate the recent user changes responsible for the problem. The need to generate explanatory messages and associate them with locations in the program text is thus avoided. (For newly inserted material, error messages can be assigned in the conventional manner.) This approach reuses existing mechanisms: the presence of errors imposes no additional requirements for persistent storage, change reporting, analysis, transformation, or editing. Tools in the environment can locate errors efficiently, and can restrict their attention to the syntactically valid structure, since unincorporated modifications are clearly identified. Since both the representation and presentation of errors are integrated with the structure and content of the program, any transformations induced by error recovery are completely reversible.

Figure 8.2 illustrates a simple example. In the initial version (V0), the program is in a consistent state. The user then modifies the program to create version V1. At this point, the structure of the program is no longer consistent with its textual content, due to the unincorporated deletion of the addition operator. In V2, the user requests that consistency be restored; incremental analysis detects the error at this time. The expression enclosing the deletion is then *isolated*, and the token containing the deletion is flagged as possessing a change that could not be successfully incorporated. The error 'message' is simply the difference in the content of the isolated region between V2 and V0 (shown as a box around the deleted operator).

The rest of this chapter is organized as follows. Section 8.2 describes the basic framework for our approach, including the representation of errors and algorithms for isolating and recording errors. In Section 8.3 we discuss techniques for incorporating additional (legal) modifications within an isolated region by retaining partially analyzed results and by applying out-of-context analysis to modified, unanalyzed subtrees. Section 8.4 considers a simple presentation scheme that

[3]Large insertions of new text, which require batch analysis and for which batch techniques represent the only possible (local) error recovery solution, may rely on language-specific information to tailor their recovery. Section 8.5.5 discusses the application of batch approaches within a contiguous region of inserted text.

combines analysis results, unincorporated material, and the contents of the distributed development log to display errors in an informative manner. Several extensions to the basic framework are covered in Section 8.5.

8.2 Modeling Errors

Chapter 4 introduced a basic program representation and a model for the editing and transformation of programs through language-specialized analysis. Here we extend that representation to include *errors*, in the form of persistent, unincorporated modifications.

8.2.1 Maintaining Unincorporated Modifications

In a program without errors, the correctness properties of the incremental lexical and syntactic analyses guarantee that the text, tokens, and structure of the program are all consistent with one another and are valid with respect to the language definition. Any modifications performed by the user introduce temporary inconsistencies among (and possibly within) these different representations. When such modifications lead to another correct program state, the next invocation of incremental analysis will transform the program representation to the new state.

When one or more modifications introduced by the user do *not* result in a syntactically correct program state, the result of incremental lexing and parsing is unspecified: the language definition and the correctness proofs of these transformations do not address the construction of a persistent representation involving errors, despite its overarching practical importance in an incremental environment. Our solution is based on an observation that is simultaneously simple and powerful: *not all modifications need to be incorporated*. We permit the inconsistency induced by one or more user modifications to persist indefinitely; the goal will be to incorporate as many valid edits as possible while leaving *all* the invalid edits unincorporated. Clearly this policy cannot violate the correctness properties of incremental lexing or parsing, as long as we consider all the unincorporated edits as pending modifications when incremental analysis is next invoked.

When only textual modifications and legal structural edits are permitted, the structure of the program representation remains well-formed (with respect to the grammar) at all times, although the lexeme⇔token mapping may be inconsistent until outstanding user modifications have been incorporated through analysis. (Section 8.5 discusses support for structural edits that violate grammatical well-formedness.) The correctness of the program structure with respect to the grammar can then be established by induction over the sequence of program transformations. There is a simple relationship between the presence of errors and consistency properties: the program text as defined by the left-to-right concatenation of the lexemes constitutes a correct program if and only if the representation is free of unincorporated modifications following the analysis.

During re-analysis of a program containing errors, the incremental lexer and parser must investigate the site of each unincorporated change, since additional modifications may have changed the surrounding context in such a way that the former error is now valid. (In the next section we describe mechanisms to limit the scope of an error.) For these and other tools in the ISDE to locate errors efficiently, each node containing an error must be distinguished, and the path between the root of the tree and each error-containing node must be marked. This is accomplished with boolean node annotations similar to the `nested` attribute provided by the history services for change reporting.[4] The incremental lexing and parsing algorithms treat error and error path annotations in the same fashion as local and nested change attributes when determining which regions of the program structure require re-analysis.

8.2.2 Isolating Errors

The drawback to the model described above is that it applies *globally*: every error site must be treated as a pending modification when re-analysis is requested, and no legal modifications can be incorporated until all the errors have been corrected. However, it is not necessary to treat the entire program as a unit; in this section we use incremental analysis and the relationship between the current analysis and the previous structure of the tree to *isolate* errors from one another.

Isolation makes our model of error recovery as unincorporated changes meaningful, by allowing legal modifications outside the isolated regions to be successfully integrated by the incremental lexer and parser. By separating the errors, isolation also improves the performance of subsequent analyses: it is not necessary to re-inspect the errors in an isolated region unless that region contains new user modifications or is affected by changes to the surrounding context. The independence of isolated structure is also useful *within* the region, since (by construction) its recovery can be computed separately from the analysis of surrounding structure or from other erroneous regions of the program.

[4]Changes to local and nested error attributes must *themselves* be captured in the history log—this allows the transformation induced by incremental analysis to be fully reversible, even when it involves error recovery.

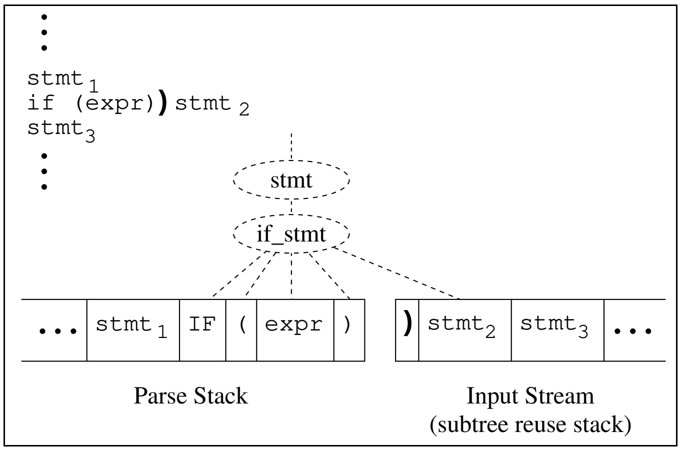


Figure 8.3: A simple isolation example ('local' recovery). The erroneous addition of an extra right parenthesis triggers the search for an isolation candidate, which succeeds immediately with the former parent of the terminal node on top of the parse stack.

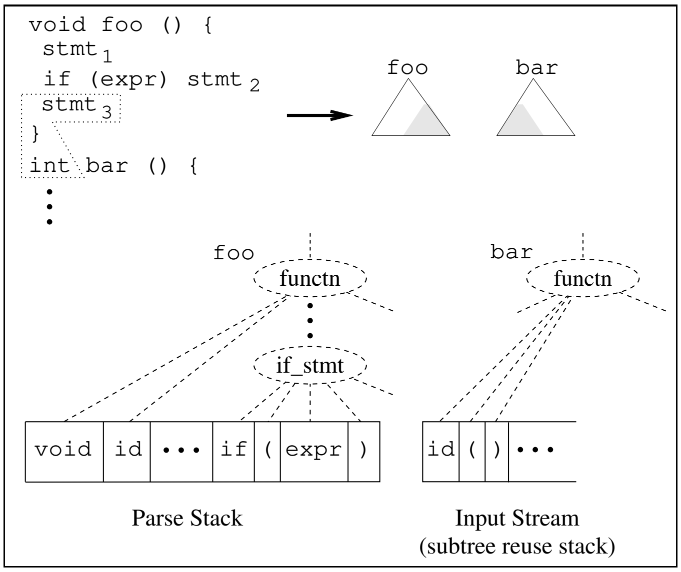


Figure 8.4: Isolation in a 'stack recovery' situation. In this case, the search for an isolation candidate requires more work; it will locate the least common ancestor of the two function nodes.

Isolation is defined by the analysis techniques: the dependencies between program components induced by lexical and syntactic analysis determine the size of an isolated region. However, computing an isolation region solely through the analysis of the current program state is problematic, since it amounts to implementing a conventional (batch) non-correcting recovery, subject to the shortcomings described in Section 8.1. Fortunately there is an additional source of information in an ISDE: the *previous* structure of the program is accessible, and can be used along with the dependency information to separate and contain errors.

Once an error has been detected by the lexer or parser, the previous structure provides a useful guide for constraining its effect. The isolation algorithm locates a well-formed subtree that existed in the previous version *and* that can be retained in the current version of the program structure to contain the site of the error. (The 'matching condition' used by some state-matching incremental parsers computes a similar relationship between the old and new trees [36, 58].) When isolated regions are small, each is likely to contain only a single error (thus preventing the recovery of one problem from contaminating another) and most correct modifications will lie outside all isolated regions (allowing them to be successfully incorporated).

Figure 8.3 contains a simple isolation example. Here the user has mistakenly inserted an additional right parenthesis following the test expression in a condition. (The syntax of C is used in this and other examples.) The problem is conceptually contained within the `if_stmt` from the previous version of the program structure. The isolation algorithm discovers this fact and 'reverts' this statement to its previous structural form. The erroneous insertion is left as an unincorporated textual modification, and presented to the user as an error by visually distinguishing the problematic character.

Isolation is not limited to purely 'local' problems. Figure 8.4 contains an example that would result in an extensive secondary repair in a conventional batch recovery. In our approach, the accidental deletion that merges the two function definitions is 'recovered' by the isolation process. The use of the previous structure allows the right side of the first function's structure and the left side of the second function's subtree to be restored. (The actual node chosen for isolation in this case will be the lowest common ancestor in the (balanced) sequence containing these function definitions. The performance implications of sequence representation for error recovery are discussed in Section 8.3.4.)

The paths to unincorporated modifications defined by `nested_error` attributes are terminated immediately below the root of the isolated subtree, preventing subsequent analyses from re-inspecting isolated errors unnecessarily.

8.2.3 Computing Isolation Regions

Figure 8.5 contains the top-level routines to initiate recovery and apply isolation. Recovery begins by removing any default reductions from the parse stack using `right_breakdown`; the associated nodes are ignored in the subsequent search for an isolation node (Figure 8.7). This search proceeds by comparing the current and previous versions of the program's structure in the region of the error. Note that it is always possible to isolate *some* subtree, since the `UltraRoot` persists across all versions. Having found a suitable candidate, the parse stack can be cut back to the beginning of the isolated subtree, the isolated subtree's root node can be shifted, the lookahead pointer can be advanced to the following subtree in the previous version, and incremental lexing/parsing can be restarted in the resulting configuration. Figure 8.2 a simple isoation example—the former parent (`expr`) of the node immediately preceding the detection point can be used to contain the error site, and parsing can successfully resume to the right of its current yield (following "`b`").

The search for an isolation candidate proceeds in two dimensions: each entry on the (current) parse stack that is not a new node is considered, and for each of these nodes its ancestors in the previous version are considered.[5] Each candidate is tested with `valid_iso_subtree`, to determine whether isolating it would cover the damaged area and be acceptable to both the lexer and parser.

The choice of an isolation region must simultaneously maintain *all* analysis invariants. The primary lexical restriction is that the isolated region contain the same text (range of offsets) in both the previous and current versions—otherwise characters would appear multiple times or be lost. (In Figure 8.2, the `expr` node can be isolated because its yield is unchanged from V1 to V2.) Additional lexical invariants may need to be imposed, depending on the expressive power of the lexical description language. (Section 8.5.3 describes the impact of several lexical description features on the isolation conditions.)

The syntactic test for a successful isolation requires that the subtree from the previous version of the program 'align' with the current parse stack—the left edge of the isolated subtree must correspond to the left edge of some subtree on the parse stack. This restriction allows the isolated subtree to replace partial analysis results by popping one or more entries off the stack and pushing the root node from the isolated subtree. Such a push must make sense with respect to the parse table: shifting the symbol (left-hand side) of the production labeling the root node of the isolated subtree must be a legal move in that configuration.

Passing the alignment and shift tests does not guarantee that the parser will be able to continue parsing successfully when the recovery is complete; the *right* context of the isolated subtree may not be legal. Although it would be possible

[5] Different search strategies could be employed; for instance, nodes deeper in the stack may sometimes be preferable to ancestors high in the previous tree.

```
setof bool paths_to_ignore;

recover () {
  paths_to_ignore = ∅;
  Find a node that exists in the previous version of the parse tree that covers the
  damaged region.
  right_breakdown();
  int sp = 0;
  Consider each node on the stack, until we reach bos.
  for (NODE *node = stack.node(); node != bos; {
    sp++;
    if (node→is_new()) continue;
    int offset = stack.offset(stack.length - 1);
    if (valid_iso_subtree(node, offset, stack.state()))
      return isolate(node, sp, 1);
    If the root of this subtree is new, keep looking down the parse stack.
    Otherwise, try searching his ancestors.
    int cut_point;
    for (NODE *ancestor = node→parent(previous_version);
         ancestor != UltraRoot && stack.get_cut(ancestor, cut_point);
         ancestor = ancestor→parent(previous_version);
      if (ancestor∉paths_to_ignore &&
          valid_iso_subtree(ancestor, stack.offset(cut_point),
                            stack.state(cut_point), cut_point))
        return isolate(ancestor, sp, cut_point);
      else {add ancestor to paths_to_ignore; node = ancestor;}
    state = stack.state(); stack.pop(); node = stack.top();
  }
  return UltraRoot; Isolate the entire tree.
}

Compute offset of leftmost character not to the left of the index th entry.
int Stack::offset (int index) {
  for (int i = 0, offset = 0; ; i++)
    if (i == index) return offset;
    Each node contains an incrementally synthesized attribute corresponding to the length,
    in characters, of its textual yield. The array entry holds the nodes on the stack.
    else offset = offset + entry[i].node→text_length(current_version);
}

Compute stack entry corresponding to leading edge of node's subtree in the previous version.
Returns false if no entry is so aligned.
bool Stack::get_cut (NODE *node, int &cut_point) {
  The value of old_offset can be computed by traversing the previous structure and
  examining the textual yields in that version.
  int old_offset = starting_offset_of_node_in_version(node, previous);
  int offset;
  for (cut_point = offset = 0; cut_point < length; cut_point++)
    if (offset > old_offset) return false;
    else if (current_offset == old_offset) return true;
    else offset = offset + entry[cut_point].node→text_length(current_version);
  return false;
}
```

Figure 8.5: Top-level error recovery. The `recover` routine begins the process by searching for an isolation candidate; the `valid_iso_subtree` test is shown in Figure 8.6. `right_breakdown` is shown in Figure 6.2; each node removed by that routine is added to `paths_to_ignore`.

```
bool valid_iso_subtree (NODE *node, int left_offset,
                        int state, int cut_point) {
  if (node∈isolation_rejects) return false;
  add node to isolation_rejects;
  The starting offset of the subtree must be the same in both the previous and
  current versions. The ending offset must meet or exceed the detection point.
  int left_offset = new_offset;
  if (left_offset != last_edited→offset(node))
    return false;
  Cannot be to the right of the point where the error was detected by the parser.
  if (left_offset > detection_offset) return false;
  if (left_offset + node→text_length(previous_version) <
       detection_offset)
    return false;
  Lexical tests—see Section 8.5.3
  Now see if the parser is willing to accept this isolation, as determined by the
  shiftability of its root symbol in the current stack configuration.
  stack.pop(cut_point);
  action = next_action(node, state);
  stack.unpop(cut_point);
  return action == SHIFT;
}
```

Figure 8.6: Procedure to test whether a given subtree is a valid isolation candidate. The tests include textual alignment with respect to the previous version of the subtree, lexical consistency checks, and an LR(0) (shift) test for the symbol labeling the root node of the subtree.

```
Perform the isolation; resets configuration so parsing can continue.
isolate (NODE *node, int sp, int cut_point) {
  stack.unpop(sp - 1);
  refine(node);
  parse_state = stack.state(sp - 1 + cut_point);
  stack.pop(sp - 1 + cut_point);
  shift(node);
  la = pop_lookahead(node);
}
```

Figure 8.7: Isolating syntax errors. Once an isolation region has been chosen by the search routine in Figure 8.5, the isolation itself is performed. This routine updates the stack configuration so that the parsing can continue immediately to the right of the isolated subtree. Figure 8.11 contains the refinement algorithm.

```
Discard changes and record errors in the subtree rooted at node;
discard_changes_and_mark_errors (NODE *node) {
  node→discard(); See Figure 3.2
  if (node→has_changes(reference_version, local)
    if (!node→local_errors) {
      node→local_errors = true;
      node→compute_presentation(reference_version);
    }
  if (∃child of node s.t.
       (child→local_errors || child→nested_errors)))
    node→nested_errors = true;
  else node→nested_errors = false;
}
```

Figure 8.8: Discarding partial analysis results and marking unincorporated modifications ('errors') local to a single node. The structure and content of the subtree rooted at `node` are reverted to their state in the previous version of the program. Any user modifications (textual or structural) within this subtree are marked as unincorporated errors, and nested error attributes are set to record the path between `node` and the location of each such error. The presentation of errors is discussed in Section 8.4.

to check this property as part of the isolation conditions, a simpler technique is to allow the parser to detect the problem and re-invoke recovery: a larger isolation region will then be selected, since all previously isolated nodes are rejected as candidates.

Once chosen, any partial results applied within an isolation region can be discovered through the usual change reporting mechanisms and removed by the algorithm in Figure 8.8; this will revert each modified subtree to its state in the previous version of the program (where the structure is known to be correct). Any user changes since the reference version are marked as unincorporated errors. By construction, modifications (valid or invalid) outside the isolated region are unaffected. (In Section 8.3 we develop methods to incorporate legal modifications *within* the isolated region.)

The general search for an isolation subtree considers only interior nodes. Since errors are sometimes contained within the textual modification(s) applied to a single token, the recovery can also consider *token-level isolation* prior to the algorithm described above. Reasonable choices of tokens to examine include the top-of-stack and lookahead symbols, as well as tokens close to them in the previous version. If a token-level isolation succeeds, the algorithm in Figure 8.8 is applied to it, the configuration is reset based on the location of the token relative to the error, and analysis is re-started.

8.2.4 Handling Lexical Errors

Although the previous sections have focused on recovery from *syntactic* errors, the mechanisms are also applicable to *lexical* problems. Errors at the lexical level can be discovered by including explicit rules in the lexical description to match invalid sequences; when one of these patterns is recognized, a special error token is created. A simpler mechanism, which can be used either alone or in concert with explicit error patterns, is *implicit* detection: instead of modifying the description, problems are discovered when characters cannot be legally recognized as belonging to any pattern; each contiguous sequence of unmatched characters produces an instance of a special *unmatched* token class. The parser will be unable to shift an unmatched-text token, since it is neither a legal whitespace token nor a terminal symbol in the grammar. The resulting error will be detected during parsing, and will trigger a recovery that handles the erroneous textual changes using the mechanisms already discussed. (The unmatched token is transient; it will be discarded during the recovery process.)

Explicit error tokens can either induce this same behavior or persist as whitespace tokens—the latter behavior is occasionally useful when the error is so common or idiosyncratic that recognizing a superset of the actual language is preferable to a normal error presentation.[6] Regardless of the policy chosen, no special effort is required to recognize, recover from, or present lexical problems. (However, recovery must respect lexical invariants as well as syntactic restrictions; Section 8.5.3 describes the impact of various features of the lexical description language on the recovery process.)

8.3 Incorporating Modifications Within Isolated Regions

If every isolated region contained only errors and no legal modifications, then applying the algorithm in Figure 8.8 to the modified portions of each isolated subtree would constitute a sufficient recovery. However, a large isolated region may contain several legal modifications (as can happen with errors involving bracketing constructs or sequences). In this section

[6] For example, some C compilers will ignore stray backslashes in the program source, except to warn the programmer of their presence.

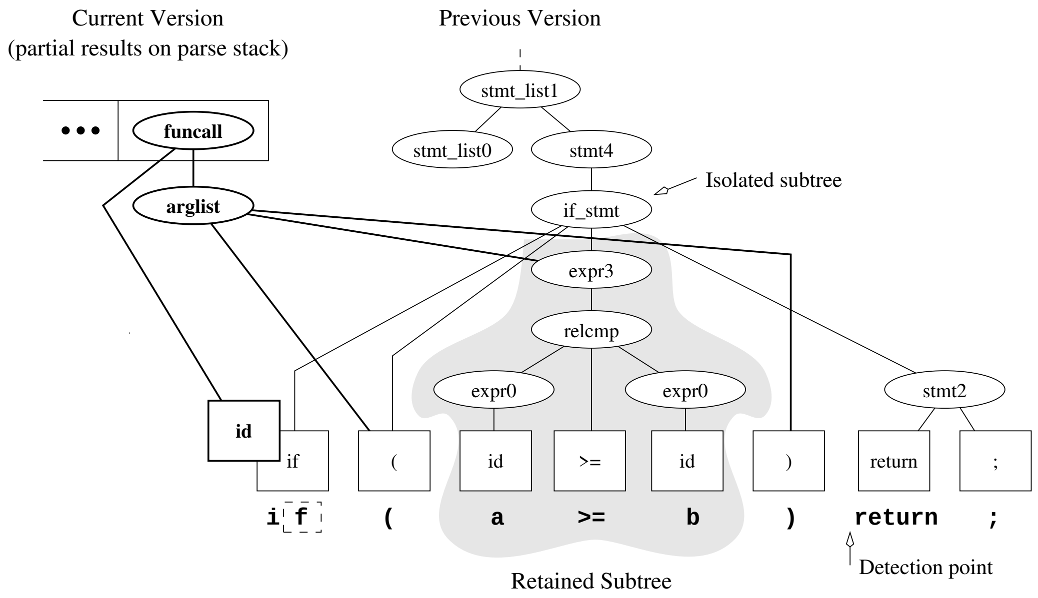


Figure 8.9: Retaining partial analysis results. The mistaken change of the keyword `if` to the identifier `i` causes the beginning of the statement to be re-interpreted as a function call. When the `return` keyword is reached, the parser detects the problem. The test expression (shaded) represents a subtree shared by *both* versions: no inspection of this subtree is required if it contains no edits. More interestingly, if it *is* changed to any other legal expression, the recovery will retain the analyzed result—the nearby error does not preclude the incorporation of correct changes to the test expression.

we consider *refinement* techniques that can integrate some, and in many cases all, of the correct modifications *within* an isolated subtree.

8.3.1 Retaining Partial Analysis Results

Refinement employs two different techniques, depending on the location of the modifications relative to the detection point that triggered the recovery. The first technique attempts to *retain* modified subtrees that have already been analyzed. When the recovery routine is invoked, the incremental lexer and parser have already seen any material to the left of the detection point. In general, several legal modifications will already have been incorporated into this material (represented by new or modified subtrees on the parse stack). Subject to certain restrictions, these subtrees can be retained instead of discarded. (Figure 8.9 illustrates a simple example.) Refinement allows *nested* isolation regions to persist in those subtrees, and avoids the redundant work of re-marking errors within them.

The two-pass retention algorithm is shown in Figure 8.10.[7] In the first pass, the previous structure of the portion of the isolated region to the left of the detection point is examined; modified subtrees that meet alignment[8] and lexical invariant restrictions can be retained in their *new* form instead of being discarded. The second pass is used to actually carry out this transformation, discarding results that cannot be retained and marking the unincorporated modifications using the algorithm of Figure 8.8. In general this process creates a 'canopy' of structure from the previous version with arbitrarily large subtrees from the current analysis embedded within it. Unmodified subtrees from the previous version that occur in the new structure are not inspected further by either pass, guaranteeing that the recovery process maintains incremental performance.

8.3.2 Out-of-Context Analysis

Retaining pre-parsed subtrees permits the incorporation of legal modifications within an isolated region to the *left* of the detection point. Modifications may also exist to the *right* of the detection point, in which case the lexer and parser will not have processed the affected subtrees. In Figure 8.9, suppose that, in addition to the error in the keyword `if`, the user replaces the return statement with a different statement. Without the techniques described below, such a modification would go unincorporated.

[7] A two-pass algorithm is necessary to avoid corrupting the new version until all the decisions about retaining portions of it have been made.

[8] Note that permitting the incorporation of valid modifications within the isolated region implies that the mapping between lexemes and characters may change, even though the character yield of the isolated region *as a whole* is unchanged from the previous version to the current one.

Given the root of a subtree from the previous version of the tree, itemize the retainable subtrees within it.

```
find_retainable_subtrees (NODE *node) {
  if (node→exists(current_version) &&
      (!node→has_changes(reference_version, nested) ||
       same_text_pos(node))) {
    add node to retainable;
    return;
  }
  foreach child of node in previous_version do
    find_retainable_subtrees(node);
}
```

Retain retainable subtrees and discard remaining structure rooted at the argument node.

```
retain_or_discard_subtrees (NODE *node, NODE *parent) {
  if (node ∈ retainable) {
    node→set_parent(parent);
    remove node from retainable;
    return;
  }
  discard_changes_and_mark_errors(node);
  foreach child of node do retain_or_discard_subtrees(node);
}
```

Figure 8.10: Computing partial analysis retention. The top function is the first pass, which computes the set of retainable subtrees. The bottom function is the second pass, which discards the effects of analysis on any regions that cannot be retained due to error recovery. `same_text_pos` determines whether a subtree's yield occupies the same character offset range as in the previous version of the program.

Even though no analysis has been performed on modified subtrees to the right of the detection point, it is not correct to simply restart analysis within the isolated region: error recovery guarantees that a legal analysis configuration has been restored only at its conclusion. Instead, we perform an *out-of-context analysis*, which attempts to analyze each (maximal) subtree containing user modifications *independent* of its surrounding context.

As with partial analysis retention, we place sufficient conditions on out-of-context analysis to ensure that any incorporated changes do not interfere with the isolation itself or with the handling of adjacent subtrees. Unlike retention, however, the sufficiency tests for out-of-context analysis are distributed: prior to analysis we verify that the target subtree contains at least one modification, has a non-null yield, and is not followed by a terminal requiring analysis. During the subtree's analysis we check whether the lexer was able to synchronize with the previous contents before reaching the subtree's right boundary—otherwise the lexical analysis might 'bleed' into the following material. Finally, at the conclusion of the subtree's analysis we must ensure that the symbol of the production labeling its (possibly changed) root is the same as in the previous version of the program.[9] When all of these conditions are met, the out-of-context analysis succeeds, and the analyzed results are integrated into the current version of the program.

The algorithms for incremental lexing and parsing during out-of-context analysis are the same as for normal analysis. To simplify the handling of out-of-context analysis, we can build a temporary set of sentinel nodes that allow the subtree to appear as the entire program (see Figure 8.12). Out-of-context parsing also requires augmenting the parse table to allow *any* symbol to serve as the start symbol [76]. A pair of distinguished terminals must be introduced for each original grammar symbol to avoid parse table conflicts: the temporary `bos` token represents the 'starting terminal', and shifting it places the parser in the correct state to process the subtree under consideration; the temporary `eos` token represents the unique termination state, and shifting it represents end of sentence for the out-of-context analysis.

The error recovery routines themselves are available during an out-of-context analysis: errors detected while the subtree is being analyzed are processed by re-entering the recovery routine. Such reentrancy permits nested isolation and refinement to occur in modified subtrees to the right of the (outer) detection point, just as they can exist in retained analysis results to the left. The failure of any sufficiency checks applied during or immediately after the out-of-context analysis of a subtree results in a nested recovery that isolates the subtree being processed. (In general, partial analysis results will be valid and will be retained within the subtree, even though the out-of-context analysis as a whole did not succeed.)

[9] Unlike Degano [21], we apply this restriction only as a mechanism for improving error recovery; this restriction does *not* apply to the user's editing model.

```
Isolate the argument and recursively recover the subtree that it roots.
refine (NODE *node) {
  int offset = last_edited→offset(node);
  pass1(node, offset);
  node→discard();
  node→local_errors = node→nested_errors = false;
  pass2(node, offset);
}

pass1 (NODE *node, int offset) {
  foreach child of node in the previous version do {
    if (offset + child→text_length(current_version) <= detection_offset)
      find_retainable_subtrees(child);
    else pass1(child, offset);
    offset += child→text_length(current_version);
  }
}

pass2 (NODE *node, int offset) {
  foreach child of node in the current version do {
    if (offset > detection_offset)
      attempt_out_of_context_analysis(child);
    else if (offset + child→text_length(current_version) <= detection_offset)
      retain_or_discard_subtrees(child, node);
    else {
      discard_changes_and_mark_errors(node);
      pass2(node, offset);
    }
    offset += child→text_length(current_version);
  }
}
```

Figure 8.11: Refining an isolated region. The `refine` routine performs two passes over the isolated subtree. The first pass is read-only and computes the set of retainable subtrees. The second pass reverts any unretainable material to the left of the detection point, invokes out-of-context analysis on any candidate subtrees to the right of the detection point, and discards changes on material that spans the detection point.

8.3.3 Refinement Algorithm

Figure 8.11 contains the top-level routine to refine the recovery of an isolated region. The isolated region is partitioned into three sections: subtrees to the left of, spanning, and to the right of the detection point. Any analysis results affecting spanning nodes are discarded (using the algorithm in Figure 8.8) during the partitioning process.

The correctness of these refinement techniques can be established easily through a left-to-right inductive proof on the subtrees within the isolation region; unmodified subtrees remain unchanged, discarded changes revert to previously correct structure, and any subtrees chosen for analysis retention or out-of-context analysis possess (by construction) sufficient conditions to ensure that their handling is independent of the surrounding material.

There are several implicit trade-offs in the computation of candidate nodes for isolation, retention, and out-of-context analysis. For isolation, increased time spent searching for a tighter isolation region may provide little practical benefit even if it succeeds, especially since the refinement techniques are so powerful. For the refinement tests, the independence constraints we impose can result in the failure to incorporate some legal changes. These restrictions could be relaxed—for instance, allowing several subtrees to be *jointly* retained or analyzed out of context, where considered singly they would fail. However, in addition to a more complicated correctness proof, looser constraints imply the need for more complex verification checks, limited backtracking, or both; any potential benefits must thus be weighed against the increased computation required and the fact that refinement as presented is already extremely effective.

8.3.4 Asymptotic Analysis

Optimal methods for incremental lexing and sentential-form parsing require time $O(t + s \lg N)$ for t new terminal symbols and s modification sites in a tree with N nodes. This result assumes that lengthy sequences are identified in the grammar and represented as balanced trees in the resulting program structure (Chapter 3 and Section 6.6). The presence of history-sensitive error recovery does not affect the running time of either algorithm, since no additional work is required until a recovery is actually invoked.

Under the same assumptions regarding the representation of lengthy sequences, the error recovery routines presented here require a worst-case running time of $O(t + s(\lg N)^2)$. The additional $\lg N$ factor is inherent in the approach: intuitively, it represents the need to compare the current and previous structure of the tree during isolation and retention. (Typical running time is likely to be closer to logarithmic, reflecting the similarity between the two versions and the fact that many common errors are local in nature. In trials with *Ensemble* using several languages, error recovery represented a negligible fraction of analysis time when measured on non-trivial programs.)

8.4 Presenting Errors

The previous sections have concentrated on detecting, isolating, and refining the program representation in the presence of errors. While an effective treatment of errors is important to the analysis algorithms and other tools in the ISDE, ultimately it is the comprehensible presentation of errors to the user that determines the effectiveness of a recovery.

Batch error recovery based on a correcting strategy typically uses information about the repair to construct an *error message* to explain the correction (and hopefully the error itself) to the user. A history-based approach provides a simpler and more effective method for communicating with the user: included among the unincorporated edits is the cause of the problem; in practice the isolation and refinement strategies are often effective in producing a set of unincorporated errors that includes all *and only* the actual errors.

In a history-based error recovery, the obvious way to present the recovery result to the user is to indicate visually the changes that were not successfully incorporated.[10] Since this interface has the user's own changes as its vocabulary and naturally correlates actual changes with the displayed indication of the problem, it subsumes and improves upon the conventional technique of generating explanatory messages.

Figures 8.1 and 8.2 suggest one way in which invalid textual insertions and deletions can be presented, using the difference between the current and the previous (correct) contents of the tokens in the affected region.[11] More elaborate presentations of the accrued changes, involving color, side-by-side comparisons, etc. can be provided using available information about the unincorporated material. The presentation attributes are computed when unsuccessful analysis results are discarded from a node; in Figure 8.8, the call

node→compute_presentation(reference_version)

indicates the computation of presentation attributes for any user changes local to `node`. The specific textual or structural changes to `node` can be extracted from its local history log. (In Figure 8.2, the arrow from V2 to V0 suggests the comparison used to determine the deleted text "+" from the affected node.)

8.5 Extensions

In this section we summarize a number of extensions to the basic history-sensitive error recovery technique.

8.5.1 Structural Editing

If the user is permitted to perform arbitrary structural editing, then the program structure is no longer guaranteed to be valid, even after an analysis is performed: consistency restoration will not be able to repair an invalid structural modification, and will be forced to leave it unincorporated. However, unincorporated structural edits represent all and only the points where the structure is invalid—the document will be 'piecewise' well-formed. Isolation, refinement, and error marking can all be extended to handle persistent invalid structure represented in this manner. (Presenting structural errors to the user is simplified by the use of a parallel structural view alongside the textual presentation of the program.)

To accommodate correct strategies for large text insertions, structural error nodes must also be permitted to possess a variable number of children, without restrictions on their types. (This is in constrast to the standard model of fixed arity, strongly-typed tree structure, but is appropriate in error situations; specially-typed sequences can be used to simulate the effect of variable arity. The number of children of a structural error node is still assumed to be effectively bounded.) The various analysis/transformation stages must be implemented in a fashion that respects the limited information content of

[10] In our experience, users do not benefit from a visual presentation of the isolation regions.

[11] The user may not correct the error by the next analysis, and may update the location containing the error without correcting the problem. Thus in composing the presentation, the recovery should combine new modifications with the existing display until the error is finally corrected (or the region is removed from the program).

such regions. Error nodes remain strongly-typed; both missing information and malformed (variable arity) structural errors can be handled as completing productions for the appropriate grammar symbol.

8.5.2 Whitespace

Explicit whitespace material[12] can be integrated into the persistent program structure of an ISDE through grammatical transformations or extensions to the incremental parser, both of which are discussed in Appendix C. Either approach can be used with the recovery techniques described here, which, with one exception, require only minor modifications to enable the 'parsing' of whitespace material during recovery.

In the representation described in Appendix C, any non-grammatical tokens are represented by connecting them to the preceding token that represents a terminal symbol in the (original) grammar. Whitespace or other non-grammatical tokens that precede the first terminal symbol are handled by treating the `bos` sentinel as if it were a part of the grammar. During an out-of-context parse, newly discovered leading whitespace will be connected to the temporary `bos` sentinel that precedes the subtree being analyzed. When that analysis completes, the temporary `bos` token will be destroyed, and any leading whitespace must be reconnected to the surrounding program structure. If the preceding terminal symbol has an existing whitespace sequence connected to it, the additional leading whitespace from the out-of-context analysis must be appended. Figure 8.12 illustrates the sequence of events.

8.5.3 Generalized Incremental Lexing

The approach to incremental lexing described in Chapter 5 includes powerful features in the lexical description language that can complicate error recovery: arbitrary lookahead, multiple start states, and atomic sequences. These features introduce the possibility of additional dependencies between tokens, imposing additional restrictions on the choice of isolation and refinement candidates. (However, inter-token dependencies are usually trivial; dependencies that arise in practice do not interfere with the power or performance of the recovery.) Here we discuss one such feature, *atomic sequences*,[13] to illustrate the effect on isolation and refinement. Support for other lexical features is similar.

In choosing both isolation candidates and subtrees for analysis retention, the leftmost and rightmost tokens must be in singular sequences (i.e., not part of any non-trivial atomic token sequence). This condition must hold in *both* the current version and previous version of the program. In testing a subtree for out-of-context analysis, this condition is checked in the previous structure only; during the out-of-context analysis an attempt by the incremental lexer to construct an atomic sequence spanning the right edge of the subtree will trigger a recursive recovery.

8.5.4 Severity Levels

The isolation and refinement methods described earlier treat all unincorporated modifications identically. However, in some cases the recovery can distinguish between modifications *known* to be errors and modifications which it cannot prove correct *or* incorrect. When a change remains unincorporated because the sufficient conditions for partial analysis retention or out-of-context analysis were not met, any unincorporated changes in the affected subtree may be valid, but analysis limitations will cause them to be treated as errors. The recovery process can expose this additional information by assigning an integer *severity level* to each unincorporated change. The interpretation of these levels will be heuristic, but an appropriate choice in their translation to presentation characteristics can assist the user in distinguishing actual problems from incomplete or insufficient analysis results.

The additional restrictions imposed by powerful lexical analysis specifications on refinement (Section 8.5.3) provide a typical application for severity levels: when lexical lookahead extends to the right of the subtree, it is unlikely that the lookahead has changed (or, if it has, that the change would affect the shape of the current subtree). In such situations, the recovery must remain conservative, but the visual display of an edit unincorporated for this reason should probably be visually distinguished from others that are almost certainly errors.

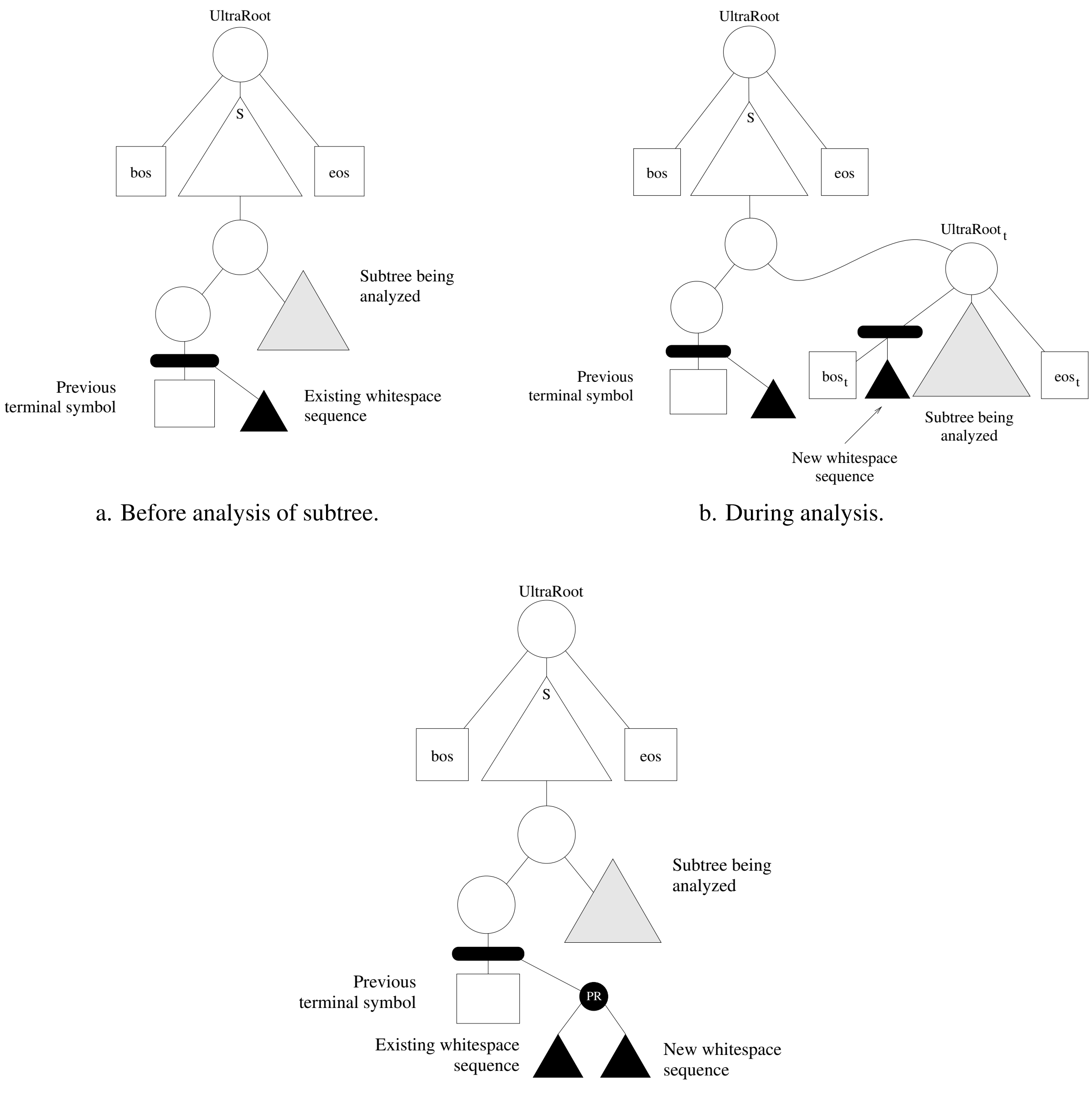
8.5.5 Recovery for Large Insertions

Insertions of large text strings mimic batch parsing, since no previous history for such material exists. Error recovery within such a region is limited to batch techniques.[14] Either correcting or non-correcting methods may be applied. (In the case

[12] We use the term generically to include any non-grammatical program material, including text-based comments.

[13] As described in Chapter 5, an atomic sequence is a contiguous range of tokens produced by a 'black box' procedure supplied by the language designer. The nature of its production renders it indivisible with respect to incrementality.

[14] Although analysis near the right edge of the region may be more powerful than in a conventional recovery, given that non-trivial subtrees are available in the input stream.



a. Before analysis of subtree.

b. During analysis.

c. After analysis.

Figure 8.12: Handling new leading whitespace created during an out-of-context analysis. In a top-level parse, leading whitespace is connected to `bos`. In an out-of-context parse, leading whitespace is connected to the temporary `bos` token until the subtree's analysis is complete. At that point, the whitespace is re-attached to the preceding terminal symbol in the surrounding tree. If a whitespace sequence is already connected to that terminal symbol (as shown here), the new sequence is merged with the existing sequence.

of a correcting strategy, persistent 'error' nodes must be introduced into the representation to model deviations from valid syntax; see Section 8.5.1 above.) Recovery methods applied to large insertions must operate only within the boundaries of the inserted material; in particular, 'stack cutting' and right context acquisition must treat material outside the inserted region as read-only. (Creation of a new program 'from scratch' is a special case of large-scale text insertion, in which the surrounding context is trivial and represented by the sentinel nodes.)

8.6 Conclusion

This chapter presents a non-correcting approach to the detection and presentation of syntactic errors that is suitable for use in an interactive and incremental software development environment. Unlike previous techniques, it uses the contents of the development log to correlate the modifications actually made by the user to errors in the program. Unlike batch non-correcting strategies, it precisely identifies the location of errors, including errors involving bracketing syntax. History-sensitive error recovery can be incorporated easily into existing algorithms for incremental lexing and parsing. The approach is itself incremental, requires no language-dependent information or user interaction, and provides a more accurate and informative report than any previous approach to error recovery.

Chapter 9

Conclusion

This dissertation makes a number of independent contributions, summarized in the conclusions of individual chapters. In addition to optimal algorithms for lexical analysis, deterministic parsing, and node reuse, we present the first known methods for incremental GLR parsing and incremental, history-sensitive error recovery. We correct prior theoretical work on persistent linked data structures, and augment this result with novel techniques for space compression to create the self-versioning document model.

Taken as a whole, our work demonstrates that incremental, language-based environments can be constructed and made to operate in a practical form on *real* languages. This is the first work to support derivation of incremental analyzers from formal descriptions for languages of commercial interest, which we regard as our central result.

A uniform document model, embracing both programs and natural language documents, can be applied profitably in an incremental environment such as *Ensemble*. Our work on self-versioning documents allows both intrinsic and imposed structure to be persistently maintained in a history log that is distributed over the document's own structure. This representation allows a novel reformulation of incremental analysis and transformation algorithms, and makes their actions—no matter how complex a restructuring is involved—fully and transparently reversible.

Finally, we hope our work will encourage commercial interest in incremental environments. By providing algorithms that use memory and computational resources efficiently and that scale to large files, modules, or classes effectively, we have shown that incremental software environments can indeed be made practical.

Bibliography

[1] Annika Aasa. Precedences in specifications and implementations of programming languages. *Theor. Comput. Sci.*, 142(1):3–26, May 1995.

[2] Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Trans. Softw. Eng. and Meth.*, 3(1):3–28, Jan. 1994.

[3] Rakesh Agrawal and Keith D. Detro. An efficient incremental LR parser for grammars with epsilon productions. *Acta Inf.*, 19:369–376, 1983.

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

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

[6] Rolf Bahlke and Gregor Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Trans. Program. Lang. Syst.*, 8(4):547–576, Oct. 1986.

[7] Robert A. Ballance, Jacob Butcher, and Susan L. Graham. Grammatical abstraction and incremental syntax analysis in a language-based editor. In *Proceedings of the ACM SIGPLAN '88 Symposium on Compiler Construction*, pages 185–198, Atlanta, Ga., Jun. 1988. ACM Press.

[8] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The Pan language-based editing system. *ACM Trans. Softw. Eng. and Meth.*, 1(1):95–127, Jan. 1992.

[9] Joseph Bates and Alon Lavie. Recognizing substrings of LR(k) languages in linear time. *ACM Trans. Program. Lang. Syst.*, 16(3):1051–1077, May 1994.

[10] John F. Beetem and Anne F. Beetem. Incremental scanning and parsing with Galaxy. *IEEE Trans. Softw. Eng.*, 17(7):641–651, Jul. 1991.

[11] U. Bianchi, P. Degano, S. Mannucci, S. Martini, B. Mojana, C. Priami, and E. Salvator. Generating the analytic component parts of syntax-directed editors with efficient error recovery. *J. Syst. Softw.*, 23(1):65–79, Oct. 1993.

[12] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 14–24, Nov. 1988.

[13] Robert Bretl et al. The GemStone data management system. In Won Kim and Federick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 283–308, 1989.

[14] Michael G. Burke and Gerald A. Fisher. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Trans. Program. Lang. Syst.*, 9(2):164–197, Apr. 1987.

[15] Martin R. Cagan. The HP SoftBench environment: An architecture for a new generation of software tools. *Hewlett-Packard Journal*, 41(3):36–47, Jun. 1990.

[16] Pehong Chen, John Coker, Michael A. Harrison, Jeffrey McCarrell, and Steven Procter. The VorTeX document preparation environment. In J. Désarménien, editor, *Second European Conference on TEX for Scientific Documentation*, pages 45–54, Strasbourg, France, Jun. 1986.

[17] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Trans. Softw. Eng.*, 16(3):325–334, Mar. 1990.

[18] Robert Corbett. `bison` release 1.24, 1992.

[19] Robert Paul Corbett. *Static Semantics and Compiler Error Recovery*. PhD dissertation, University of California, Berkeley, 1985. Available as technical report UCB/CSD–85–251.

[20] Gordon V. Cormack. An LR substring parser for noncorrecting syntax error recovery. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, volume 24(7), pages 161–169. ACM Press, Jun. 1989.

[21] Pierpaolo Degano, Stefano Mannucci, and Bruno Mojana. Efficient incremental LR parsing for syntax-directed editors. *ACM Trans. Program. Lang. Syst.*, 10(3):345–373, Jul. 1988.

[22] Pierpaolo Degano and Corrado Priami. Comparison of syntactic error handling in LR parsers. *Software—Practice & Experience*, 25(6):657–679, Jun. 1995.

[23] Norman M. Delisle, David E. Menicosy, and Mayer D. Schwartz. Viewing a programming environment as a single tool. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 49–56. ACM Press, 1986.

[24] Brian M. Dennis. ExL: The Ensemble extension language. Master's thesis, Computer Science Division—EECS, University of California, Berkeley, May 1994.

[25] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symp. on Theory of Computing*, pages 365–372, 1987.

[26] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, Feb. 1989.

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

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

[29] P. Feiler, S. Dart, and G. Downey. Evaluation of the Rational environment. Technical Report CMU/SEI-88-TR-15, Software Engineering Institute, Carnegie-Mellon University, 1988.

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

[31] Bernd Fischer, Carsten Hammer, and Werner Struckmann. ALADIN: A scanner generator for incremental programming environments. *Software—Practice & Experience*, 22(11):1011–1025, 1992.

[32] C. N. Fischer, Gregory F. Johnson, John Mauney, Anil Pal, and Daniel L. Stock. The Poe languages-based editor project. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*. ACM Press, 1986.

[33] Francois Fluckiger. *Understanding Distributed Multimedia: Applications and Technology*. Prentice-Hall, Englewood Cliffs, N.J., 1996.

[34] Christopher W. Fraser and Eugene W. Myers. An editor for revision control. *Software—Practice & Experience*, 9(2):277–295, 1987.

[35] Neal M. Gafter. *Parallel Incremental Compilation*. PhD dissertation, University of Rochester, Rochester, N.Y., 1990.

[36] Carlo Ghezzi and Dino Mandrioli. Augmenting parsers to support incrementality. *Journal of the ACM*, 27(3):564–579, Jul. 1980.

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

[38] Susan L. Graham, C. B. Haley, and W. N. Joy. Practical LR error recovery. In *Proc. SIGPLAN 79 Symposium on Compiler Construction*, pages 168–175, DenverCol., Aug. 1979. ACM Press.

[39] Susan L. Graham, Michael A. Harrison, and Ethan V. Munson. The Proteus presentation system. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages 130–138. ACM Press, Dec. 1992.

[40] Michael A. Harrison and Vance Maverick. Presentation by tree transformation. In *CompCon '97*, pages 68–73, San Jose, Calif., Feb. 1997. IEEE Computer Society Press.

[41] Görel Hedin. *Incremental Semantic Analysis*. PhD dissertation, Department of Computer Science, Lund University, Mar. 1992.

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

[43] J. Heering, P. Klint, and J. Rekers. Incremental generation of lexical scanners. *ACM Trans. Program. Lang. Syst.*, 14(4):490–520, Oct. 1992.

[44] Stephan Heilbrunner. On the definition of elr(k) and ell(k) grammars. *Acta Inf.*, 11:169–176, 1979.

[45] Susan B. Horwitz. Generating language-based editors: A relationally-attributed approach. Technical Report TR 85-696, Department of Computer Science, Cornell University, 1985.

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

[47] Scott E. Hudson and Roger King. The Cactis project: Database support for software environments. *IEEE Trans. Softw. Eng.*, 14(6):709–719, Jun. 1988.

[48] Ian Jacobs and Laurence Rideau-Gallot. A Centaur tutorial. Technical Report 140, INRIA, Jul. 1992.

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

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

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

[52] Randy H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Comput. Surv.*, 22(4):275–408, 1990.

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

[54] J. L. Knudsen, M. Löfgren, O. L. Madsen, and B. Magnusson. *Object-Oriented Environments — The Mjølner Approach*. Prentice-Hall, 1993.

[55] Donald E. Knuth. *The TeXbook*, volume A of *Computers and Typesetting*. Addison-Wesley, Reading, Mass., 1986.

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

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

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

[59] M. E. Lesk and E. Schmidt. LEX—a lexical analyzer generator. In *Unix Programmer's Manual*. Bell Telephone Laboratories, 7th edition, Jan. 1979.

[60] Warren Xiaohui Li. A simple and efficient incremental LL(1) parsing [sic]. In *SOFSEM '95: Theory and Practice of Informatics*, LNCS, pages 399–404, Berlin, November/December 1995. Springer-Verlag.

[61] Warren Xiaohui Li. *Towards Generating Practical Language-Based Editing Systems*. PhD dissertation, University of Western Australia, 1995.

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

[63] William Maddox. *Incremental Static Semantic Analysis*. PhD dissertation, University of California, Berkeley, 1997. Available as technical report UCB/CSD–97–948.

[64] Boris Magnusson, Ulf Asklund, and Sten Minör. Fine-grained revision control for collaborative software development. In *First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 33–41. ACM Press, 1993.

[65] Vance Maverick. *Presentation by Tree Transformation*. PhD dissertation, University of California, Berkeley, 1997. Available as technical report UCB/CSD–97–947.

[66] Raul Medina-Mora. Syntax directed editing: Towards integrated programming environments. Technical Report CMU-CS-81-113, Department of Computer Science, Carnegie-Mellon University, March 1982. Ph.D. dissertation.

[67] B. Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1992.

[68] Microsoft Corp., Inc. Visual C++, 1997.

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

[70] Ethan V. Munson. *The Proteus Presentation System*. PhD dissertation, University of California, Berkeley, 1992. Available as technical report UCB/CSD–94–833.

[71] Arvind M. Murching, Y. V. Prasad, and Y. N. Srikant. Incremental recursive descent parsing. *Computer Languages*, 15(4):193–204, 1990.

[72] D. Notkin, R. J. Ellison, B. J. Staudt, G. E. Kaiser, E. Kant, A. N. Habermann, V. Ambriola, and C. Montangero. Special issue on the GANDALF project. *J. Syst. Softw.*, 5(2), May 1985.

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

[74] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, Mass., 1994.

[75] Vern Paxson. `flex 2.5.2` man pages, Nov. 1995. Free Software Foundation.

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

[77] PROCASE Corporation, 2694 Orchard Parkway, San Jose, California 95134. *SMARTsystem™ Reference Guide*, release 2.0 edition, Mar. 1993.

[78] Vincent Quint and Irène Vatton. Grif: An interactive system for structured document manipulation. In J. C. van Vliet, editor, *Text processing and document manipulation*, pages 200–213. Cambridge University Press, Apr. 1986.

[79] Jan Rekers. *Parser Generation for Interactive Environments*. PhD dissertation, University of Amsterdam, 1992.

[80] Jan Rekers and Wilco Koorn. Substring parsing for arbitrary context-free grammars. *SIGPLAN Not.*, 26(5):59–66, May 1991.

[81] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, Berlin, 1989.

[82] H. Richter. Noncorrecting syntax error recovery. *ACM Trans. Program. Lang. Syst.*, 7(3):478–489, Jul. 1985.

[83] Graham Ross. Integral C – A practical environment for C programming. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 42–48. ACM Press, 1986.

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

[85] Masataka Sassa and Ikuo Nakata. A simple realization of LR-parsers for regular right part grammars. *Information Processing Letters*, 24:113–120, Jan. 1987.

[86] John J. Shilling. Incremental LL(1) parsing in language-based editors. *IEEE Trans. Softw. Eng.*, 19(9):935–940, Sep. 1992.

[87] E. Steegmans, J. Lewi, and I. Van Horebeek. Generation of interactive parsers with error handling. *IEEE Trans. Softw. Eng.*, 18(5):357–367, May 1992.

[88] Duane Szafron and Randy Ng. LexAGen: An interactive incremental scanner generator. *Software—Practice & Experience*, 20(5):459–483, May 1990.

[89] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, Penn., 1983.

[90] Mikkel Thorup. Controlled grammatic ambiguity. *ACM Trans. Program. Lang. Syst.*, 16(3):1024–1050, May 1994.

[91] Walter F. Tichy. RCS — a system for version control. *Software—Practice & Experience*, 15(7):637–654, July 1985.

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

[93] Michael L. Van De Vanter, Susan L. Graham, and Robert A. Ballance. Coherent user interfaces for language-based editing systems. *Intl. J. Man-Machine Studies*, 37:431–466, 1992.

[94] Mark van den Brand and Eelco Visser. Generation of formatters for context-free languages. *ACM Trans. Softw. Eng. and Meth.*, 5(1):1–41, Jan. 1996.

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

[96] Eelco Visser. A family of syntax definition formalisms. In *Proceedings of ASF+SDF95—A Workshop on Generating Tools from Algebraic Specifications*, May 1995.

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

[98] Tim A. Wagner and Susan L. Graham. Integrating incremental analysis with version management. In *5th European Softw. Eng. Conf.*, number 989 in LNCS, pages 205–218, Berlin, Sep. 1995. Springer-Verlag.

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

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

[101] Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, page To appear. ACM Press, Jun. 1997.

[102] W. M. Waite. The cost of lexical analysis. *Software—Practice & Experience*, 16(5):473–488, May 1986.

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

[104] Mark N. Wegman. Parsing for structural editors. In *Proc. of 21st Annual IEEE Symposium on Foundations of Computer Science*, pages 320–327, Syracuse, N.Y., Oct. 1980. IEEE Press.

[105] Wuu Yang. Incremental LR parsing. In *1994 International Computer Symposium Conference Proceedings*, pages 577–583 vol. 1, National Chiao Tung University, Hsinchu, Taiwan, 1994.

[106] Dashing Yeh and Uwe Kastens. Automatic construction of incremental LR(1) parsers. *ACM SIGPLAN Notices*, 23(3):33–42, Mar. 1988.

Appendix A

Versioning-Related Algorithms

The two algorithms in this Appendix are part of the implementation of self-versioning documents described in Chapter 3. The algorithms are computationally similar, though they differ in some crucial details.

A.1 Version Alteration Algorithm for Objects with Differential Storage

The following algorithm is used to alter the current (cached) version of a differential versioned object. Conceptually, it forms the path in the global version tree between the current version and the target version, then projects this path onto its local equivalent. The elements of the local path are exactly the differential log entries that must be applied (in the proper order) to transform the current version of the object to its value in the target version. Neither path is constructed explicitly; the actual computation is traversal-based. Note that some unnecessary projections have been optimized out by directly updating the global version identifier.

The right ('redo') side of the path is computed recursively, starting from the target version. The actual update takes place as the recursion unwinds.

```
void diff_vobject::alter_version (const VG *vg, VData *vd, int target_gvid) {
  Vlstring *vls = (Vlstring*)vd;
  int from = vg→currentI;
  int   to = target_gvid;

  if (from == to) return;

  Updated current pointer in the data object before we begin.
  vls→current = project(vg, target_gvid);

  GD *from_gd = vg→log→maps[from];
  GD *to_gd   = vg→log→maps[to  ];

  This is both the returned value from project()  and the
  limit setting for the subsequent project() call.
  int lvid = -1;

  if (from_gd == to_gd) {
    Common (and easy!) case: source and target in same linear run.
    if (from > to) {
      for (lvid = project(vg, from); gvids[lvid] > to; --lvid)
        vls→apply(values[lvid], false);
    } else if (to > from) {
      for (lvid = project(vg, from + 1);
           lvid < num && gvids[lvid] > from && gvids[lvid] <= to;
           lvid++)
        vls→apply(values[lvid], true);
    }
    return;
  }
```

```
  else {
    Uncommon (and hard) case: perform a general search.
    const GD *gd = from_gd;
    if (vg→log→ancestor(gd, to_gd) && from + 1 < vg→log→num_maps) {
      Handle suffix in starting gd.
      for (lvid = project(vg, from + 1);
           lvid < num && gvids[lvid] > from &&
             gvids[lvid] < gd→start + gd→num;
           lvid++)
        vls→apply(values[lvid], true);
    } else {
      Handle prefix in starting gd.
      for (lvid = project(vg, from);
           gvids[lvid] >= from_gd→start &&
             gvids[lvid] < from_gd→start + from_gd→num;
           --lvid)
        vls→apply(values[lvid], false);
      bool need_proj = true;
      gd = from_gd→parent;
      while (!vg→log→ancestor(gd, to_gd)) {
        if (need_proj) lvid = project(vg, gd→start + gd→num - 1, lvid);
        for (;
             gvids[lvid] >= gd→start && gvids[lvid] < gd→start + gd→num;
             --lvid)
          vls→apply(values[lvid], false);
        need_proj = gd→parent→kids[0] != gd;
        gd = gd→parent;
      }
    }
    const GD *lca = gd;
    Now we process the redo side.
    This is not handled the same way, since the path must be traversed top-to-bottom, and we
    cannot discover it that way. Use recursion to wind up the path and then traverse the path
    during the unwind. The LCA has already been computed.
    gd = to_gd;
    if (vg→log→ancestor(gd, from_gd)) {
      Handle suffix in ending gd.
      for (lvid = project(vg, gd→start + gd→num - 1);
           gvids[lvid] > to;
           --lvid)
        vls→apply(values[lvid], false);
    }
    else {
      int lvid = project(vg, to_gd→start);
      int lvid_copy = lvid;
      find_path_and_unwind_doing_alter_version(vg, vls, gd, lca, lvid);
      Handle prefix in ending gd.
      for (lvid = lvid_copy;
           lvid < num && gvids[lvid] >= to_gd→start && gvids[lvid] <= to;
           lvid++)
        vls→apply(values[lvid], true);
    }
  }
}
```

```
void find_path_and_unwind_doing_alter_version (Vlstring *vls,
                                               const GD *gd,
                                               const GD *lca,
                                               int &lvid) {
  const GD *parent = gd→parent;
  if (parent == lca) return;
  lvid = project(vg, parent→start, lvid);
  int local_lvid = lvid;
  find_path_and_unwind_doing_alter_version(vg, vls, parent, lca, lvid);
  for (;
       lvid < num && gvids[lvid] >= parent→start &&
         gvids[lvid] < parent→start + parent→num;
       lvid++)
    vls→apply(values[lvid], true);
}
```

A.2 Computation of Nested Changes

The determination of nested change values during a specific interval is computationally similar to the previous algorithm, although recursion is unnecessary in this case due to the state-based representation of boolean values. The treatment of the lowest common ancestor (LCA) in the global version tree and the handling of boundary conditions is also slightly different, due to the fact that this algorithm deals with a disjunction of *transition* values during a time period, not values *per se*. (In this respect nested changes are unlike other versioned booleans, although they are represented in the same manner.)

```
bool nested_change_bit::true_during (const GVID  from_gvid,
                                     const GVID  to_gvid) const {
  int from = from_gvid.cell;
  int   to =   to_gvid.cell;
  assert(from >= 0 && from < vg→log→num_maps);
  assert(to   >= 0 && to   < vg→log→num_maps);

  if (from == to) return false;

  GD *from_gd = vg→log→maps[from];
  GD *to_gd   = vg→log→maps[to  ];

  if (IS_TMP) {
    if (value != TMP→value) return false;
    if (vg→log→ancestor(to, from)) return exists(vg, from_gvid);
    else if (vg→log→ancestor(from, to)) return exists(vg, to_gvid);
    else return exists(vg, to_gvid) || exists(vg, from_gvid);
  } else {
    This is both the returned value from project() and the limit setting for the
    subsequent project() call.
    int lvid = -1;

    if (from_gd == to_gd) {
      Common (and easy!) case: source and target in same linear run.
      if (from > to) {
        for (lvid = LOG→project(vg, from); LOG→gvids[lvid] > to; lvid--)
          if ((LOG→states[lvid] == NORMAL || LOG→states[lvid] == TEMPORARY) &&
              GET_BV(lvid) == value) return true;
        The following handles the LCA problem.
        lvid = LOG→project(vg, to + 1);
        if ((LOG→states[lvid] == NORMAL || LOG→states[lvid] == TEMPORARY) &&
            GET_BV(lvid) == value) return true;
      } else if (to > from) {
        for (lvid = LOG→project(vg, from + 1);
             lvid < LOG→num &&
               LOG→gvids[lvid] > from && LOG→gvids[lvid] <= to;
             ++lvid)
          if ((LOG→states[lvid] == NORMAL || LOG→states[lvid] == TEMPORARY) &&
              GET_BV(lvid) == value) return true;
        The following handles the LCA problem.
        lvid = LOG→project(vg, from + 1);
        if ((LOG→states[lvid] == NORMAL || LOG→states[lvid] == TEMPORARY) &&
            GET_BV(lvid) == value) return true;
      }
    }
```

```
else {
  Uncommon (and hard) case: perform a general search.
  const GD *gd = from_gd;
  if (vg→log→ancestor(gd, to_gd) && from + 1 < vg→log→num_maps) {
    Handle suffix in starting gd.
    for (lvid = LOG→project(vg, from + 1);
          lvid < LOG→num && LOG→gvids[lvid] > from &&
            LOG→gvids[lvid] < gd→start + gd→num;
          ++lvid)
      if ((LOG→states[lvid] == NORMAL || LOG→states[lvid] == TEMPORARY) &&
           GET_BV(lvid) == value) return true;
    The LCA problem: either the penultimate gvid is inside 'gd', in which case we processed it in the above loop, or
    it's the starting gvid in the child that's going to be the penultimate gd on the redo path, which we take care
    of below.
  } else {
    Handle prefix in starting gd.
    for (lvid = LOG→project(vg, from);
          LOG→gvids[lvid] >= from_gd→start &&
            LOG→gvids[lvid] < from_gd→start + from_gd→num;
          lvid--)
      if ((LOG→states[lvid] == NORMAL || LOG→states[lvid] == TEMPORARY) &&
           GET_BV(lvid) == value) return true;
    bool need_proj = true;
    const GD *last_gd = gd;
    gd = from_gd→parent;
    while (!vg→log→ancestor(gd, to_gd)) {
      if (need_proj) lvid = LOG→project(vg, gd→start + gd→num - 1, lvid);
      for (;
            LOG→gvids[lvid] >= gd→start &&
              LOG→gvids[lvid] < gd→start + gd→num;
            lvid--)
        if ((LOG→states[lvid] == NORMAL ||
              LOG→states[lvid] == TEMPORARY) &&
             GET_BV(lvid) == value) return true;
      need_proj = gd→parent→kids[0] != gd;
      last_gd = gd;
      gd = gd→parent;
    }
    The penultimate gvid on the undo side, if it's not inside a GD, must be the starting gvid of the penultimate GD on this
    path. So project that explicitly.
    lvid = LOG→project(vg, last_gd→start);
    if ((LOG→states[lvid] == NORMAL || LOG→states[lvid] == TEMPORARY) &&
        GET_BV(lvid) == value) return true;
  }
  const GD *lca = gd;
  Now we process the redo side in a symmetric fashion. The only difference is that we don't need to do ancestor
  computations, since we already know what the LCA is.
  gd = to_gd;
  if (vg→log→ancestor(gd, from_gd) && to + 1 < vg→log→num_maps) {
    Handle suffix in ending gd.
    for (lvid = LOG→project(vg, to + 1);
          lvid < LOG→num && LOG→gvids[lvid] < gd→start + gd→num &&
            LOG→gvids[lvid] > to;
          ++lvid)
      if ((LOG→states[lvid] == NORMAL || LOG→states[lvid] == TEMPORARY) &&
           GET_BV(lvid) == value) return true;
    The LCA problem: either the penultimate gvid is inside 'gd', in which case we processed it in the above loop, or
    it's the starting gvid in the child that's going from be the penultimate gd on the redo path, which we take care
    of below.
  }
```

```
      else {
        Handle prefix in ending gd.
        for (lvid = LOG→project(vg, to);
             LOG→gvids[lvid] >= to_gd→start &&
               LOG→gvids[lvid] < to_gd→start + to_gd→num;
             lvid--)
          if ((LOG→states[lvid] == NORMAL || LOG→states[lvid] == TEMPORARY) &&
              GET_BV(lvid) == value) return true;
        bool need_proj = true;
        const GD *last_gd = gd;
        gd = to_gd→parent;
        while (gd != lca) {
          if (need_proj) lvid = LOG→project(vg, gd→start + gd→num - 1, lvid);
          for (;
               LOG→gvids[lvid] >= gd→start &&
                 LOG→gvids[lvid] < gd→start + gd→num;
               lvid--)
            if ((LOG→states[lvid] == NORMAL ||
                 LOG→states[lvid] == TEMPORARY) &&
                GET_BV(lvid) == value) return true;
          need_proj = gd→parent→kids[0] != gd;
          last_gd = gd;
          gd = gd→parent;
        }
        The penultimate gvid on the redo side, if it's not inside a GD, must be the starting gvid of the penultimate GD on this
        path.  So project that explicitly.
        lvid = LOG→project(vg, last_gd→start);
        if ((LOG→states[lvid] == NORMAL || LOG→states[lvid] == TEMPORARY) &&
            GET_BV(lvid) == value) return true;
      }
    }
  }
  No matching value found.
  return false;
}
```

Appendix B

IGLR Parsing Algorithm

The non-deterministic component of the IGLR parser is based on Rekers' batch parser [79]. The routines `left_breakdown` and `pop_lookahead` are described in Chapter 6.

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

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

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

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

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

Main routine

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

reduce shift sequence*

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

Transition one parser

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

Shift all parsers

```
shifter () {
  if (is_terminal(shiftLa) && shiftLa→has_changes(lastParsedVersion))
    relex(shiftLa); Invoke lexer and reset lookaheads
  activeParsers = ∅;
  multipleStates = |for_shifter| > 1;
  while (!is_term(shiftLa) && (multipleStates ||
         forShifter→state ≠ shiftLa→state))
    shiftLa = left_breakdown(shiftLa);
  Consider all the state/link head pairs in forShifter.
  ∀<state,gss> ∈ forShifter do
    if (∃p ∈ activeParsers with p→state == state)
      p→add_link(new LINK(gss, shiftLa));
    else activeParsers = activeParser ∪ new GSS_NODE(state, new LINK(gss, shiftLa))
}
```

Find all paths

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

Path-restricted version of above function

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

Perform a single reduction

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

Get offset range

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

Create or reuse a ‘production’ node

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

Instantiate symbol nodes lazily

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

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

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

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

Note that this approach is conservative—it invalidates more structure than the optimistic sentential-form parser of Chapter 6, even when the parse table, program, and modification history are identical.

Appendix C

Modeling User-Provided Whitespace and Comments

The handling of whitespace in batch environments is well understood for a variety of programming languages and whitespace models. However, *incremental* environments have not provided a satisfactory solution to this problem, due to the fact that this material must be included (and incrementally maintained) in the persistent representation, rather than simply discarded. We describe a combination of representation and analysis techniques that together provide generic support for explicit whitespace and similar 'extra-grammatical' material in an ISDE. Our methods are independent of the lexical model, handling whitespace-insensitive languages (such as Fortran77), whitespace-sensitive languages (such as C++), and mixed-sensitivity languages (such as Haskell). The representation is uniformly structural: whitespace is integrated with the persistent program structure without introducing special cases. The representation is also efficient, imposing minimal overhead and guaranteeing logarithmic access times to *all* nodes. Two simple strategies for incrementally maintaining the proposed structure are described, one based on grammar transformation and the other on modifying the incremental parser.

C.1 Introduction

Whitespace plays several roles in programming languages: it separates other tokens, provides the programmer with a degree of control over the visual presentation of the source code, and in some languages even serves as a syntactic construct. Batch compilers and environments handle whitespace in a simple manner, discarding it as the lexical analyzer scans each region of the text.[1] *Incremental* environments, however, have typically failed to provide an adequate solution for handling whitespace [6, 8, 81]. When these environments are also multilingual, the need to simultaneously support multiple whitespace models exacerbates the problem. The lack of incremental, language-independent support for whitespace has limited both the scope and functionality of these environments.

One reason for this limitation is that the program representation in an incremental environment is both *persistent* and *structural*, requiring a different handling of whitespace than the simple approach taken in batch environments. Also, many interactive environments support high-quality program presentation services, where layout can be derived rather than forcing the programmer to express it solely through whitespace characters embedded in the program content. However, even in this setting, the programmer must be permitted to override the standard layout in order to express semantically significant layout decisions in a persistent fashion, requiring implicit and explicit whitespace to coexist.

In addition to the use of whitespace to influence the visual appearance of the program, the program representation must typically support many other elements that stand in a similar relation to the normal program structure: text-based comments,[2] constructs from embedded languages, transient representations of edited text (such as inserted text not yet incorporated by incremental analysis), and 'errors' in the form of material that was not successfully incorporated into the program structure during some previous incremental analysis. Supporting *any* of these elements requires the environment to address the same representation and reconstruction issues as with explicit whitespace.

In this appendix we describe a simple integration of whitespace material with the persistent structural representation of the program. Our representation is independent of the whitespace model, and therefore of the language itself. The uniformity of the integration provides significant leverage: existing tools can easily view or ignore whitespace without introducing special cases, both structural and textual views of the program are supported, and all editing, change reporting, versioning,

[1] Languages in which whitespace can play a syntactic role naturally require additional communication between the lexer and parser in some cases.

[2] The need for environments to support existing programs and to interact with external tools will require them to support text-based comments in addition to structural annotations for the foreseeable future.

and analysis/transformation services treat whitespace material in the same manner as other program components. Our representation is also efficient, imposing minimal space overhead and guaranteeing logarithmic access times to all whitespace tokens by representing each contiguous sequence as a balanced binary tree.

By specifying only the structure of the program's representation, this method is independent of the lexical specification and analysis mechanisms that describe and create whitespace tokens. It is also compatible with a wide variety of presentation services and approaches: layout can be based solely on whitespace material embedded in the source text, completely independent of it, or use some combination of the two methods [94].

Unlike batch systems, which rely primarily on the *lexer* to handle whitespace (where it is discarded once recognized), incremental systems use the incremental *parser* to integrate this material into the persistent program structure. We describe two methods for constructing our representation. Both support declarative language definition, an unrestricted editing model, and highly efficient incremental reconstruction of the program representation to incorporate the user's modifications.

The first method is based on grammar transformation. The conversion of the original grammar to a canonical form (Section 6.6) can make assumptions about the existence of whitespace explicit: the transformed grammar will include a (possibly empty) sequence of whitespace tokens between terminals in the source grammar. Common classes of grammars used for deterministic parsing (LALR(1), LL(1)) are closed under this transformation. By permitting whitespace to be mentioned in the *original* grammar, this approach can also be used with languages like Haskell, which allow whitespace to play a syntactic role in certain contexts.

The second approach encodes the additional instructions needed to build the whitespace representation directly into the incremental parsing algorithm. Existing incremental parsing algorithms based on either state-matching or sentential-form parsing can be extended easily to construct this representation without compromising incremental performance.

C.2 Integrating Whitespace with the Program Structure

We describe enhancements to the program representation developed in Part I to allow whitespace, comments appearing in the program text, and similar material to be present. Hereafter, the term 'whitespace' is used to describe *any* material logically present in the token stream but not regarded as terminal symbols in the grammar. (The treatment of mixed-model languages, such as Haskell, is deferred until Section C.3.)

A persistent representation of whitespace elements in an ISDE must fulfill a number of requirements:

Ordering: Whitespace must appear correctly ordered with respect to other program elements.

Uniformity: Clients and services such as editing, change reporting, and incremental analysis should have a single method for viewing the structure and content of programs.

Efficiency: There should be no limit on the number of contiguous whitespace tokens, and clients should be able to access each token in time logarithmic in the length of the sequence.

Abstraction: Clients should be able to view the program (or any subset of it) with or without whitespace, and both views should be equally efficient to produce.

Flexibility: The representation should be independent of the language, the whitespace model, the number and types of whitespace categories, and the mechanisms by which they are specified and discovered during lexical analysis.

The implementation is straightforward: each contiguous sequence of whitespace tokens is represented as a balanced binary tree. (Often this 'tree' will consist of a single whitespace token.) The sequence is logically associated with the preceding token that represents an instance of a terminal symbol in the grammar. An additional *connector node* is introduced to serve as the parent of the both the preceding terminal and the root of the whitespace sequence. Figure C.1 illustrates the arrangement. Whitespace at the beginning of the program follows the `bos` sentinel token (Figure 3.2), maintaining uniformity in the representation.[3]

This approach meets all the criteria described above. Ordering is obviously maintained; the original program text can be reconstituted simply by walking the tree, inspecting the content of each lexeme encountered. Clients can navigate whitespace sequences with the same structural operations used elsewhere in the program, and whitespace tokens are implemented in the same fashion as other nodes.

[3]The use of out-of-context parsing for error recovery requires the ability to re-attach and merge whitespace sequences, which is supported effectively by the representation described here. Section 8.5.2 discusses the interaction between out-of-context analysis and the presence of extra-grammatical tokens in more detail.

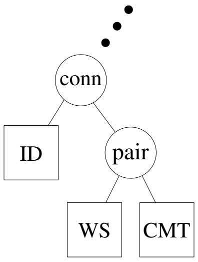


Figure C.1: Structural representation of whitespace material. A contiguous sequence of extra-grammatical tokens, which can include whitespace, text-based comments, unincorporated text, errors, and similar elements, are grouped and connected to the preceding token that represents a terminal symbol in the grammar.

```
/* This is comment line 1 */
/* This is comment line 2 */
           .
           .
           .
/* This is comment line n */
```

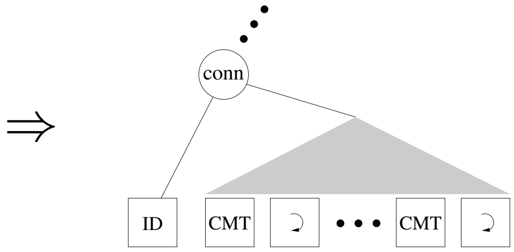


Figure C.2: The representation of contiguous whitespace tokens. To prevent long sequences from degrading performance, a balanced binary tree is used to guarantee logarithmic access time to each token. (Curly arrows represent newlines.)

As with other potentially unbounded sequences, a contiguous sequence of whitespace tokens must provide efficient navigation and traversal regardless of its length. Since these sequences are associative by definition, a balanced representation can be employed (Chapter 3, Section 6.6). Although the average number of whitespace tokens following a given terminal symbol is usually small, whitespace sequences can become quite lengthy, as Figure C.2 illustrates.[4]

In our environment, most tools view the structural representation through an *abstraction* that tailors the tree view for their particular needs. Analysis tools, including the incremental lexer and parser, use abstraction in order to access versions of the program other than the current one. Other tools, such as presentation, use abstraction to customize their view of the program content. Abstractions are implemented in a stateless 'navigational' style.

The visibility of whitespace material is also handled by the abstraction mechanism. Since all nodes—including whitespace nodes—are typed, an abstraction can easily filter out unwanted material based on the node type. In the particular case of whitespace, tools that want to eliminate whitespace can do so by using an abstraction that treats each connector node as a proxy for its first child. Figures C.3 through C.5 illustrate both textual and structural views of a sample program under different abstractions.

The presence of whitespace does not affect the editing model. Text inserted between existing tokens can be represented as an extension to one of the tokens or as a new 'unincorporated insertion' token placed between them and represented in the same fashion as whitespace tokens.[5]

When multiple categories of whitespace tokens exist, clients may require an efficient access path to the subset of tokens from a particular category within a heterogeneous whitespace sequence. To make this filtering process efficient, we can annotate each node in the binary tree representing the whitespace sequence with information that summarizes the union of the categories of the tokens in its yield. (Equivalently, the namespace of node types can be extended to convey this information, with the appropriate type selected whenever a structural editing operation occurs.) As with other summary information about the descendants of a node, category information can be incrementally maintained as the program structure is modified during editing, incremental parsing, or other transformations.

[4] Since the token is the unit of granularity for both re-analysis and representation, the language description writer must take care to ensure that lexemes are short in practice. For instance, if whitespace or textual comments in a particular language typically span many lines, each line may be described as a separate token in order to increase the effectiveness of balancing.

[5] One complication in a completely whitespace-insensitive language is that whitespace elements can occur *within* an otherwise-atomic lexeme. This problem is most easily handled by providing two distinct views for each token's lexeme: a *verbatim* view appropriate for re-creating the exact textual content of the program and a *filtered* view that represents the lexeme as seen by most clients, with any whitespace characters removed.

```
assign →  id = exp ;
  expr →  expr && expr
       |  expr —— expr
       |  id
       |  . . .
  . . .
```

```
x␣=␣a&&b||
    c&&d;
```

```
x=a&&b||c&&d;
```

Figure C.3: Running example. From left to right: grammar, program text including whitespace, program text without whitespace. (Curly arrows represent newlines, horizontal arrows denote tabs.) Figures C.4 and C.5 illustrate structural views of this example.

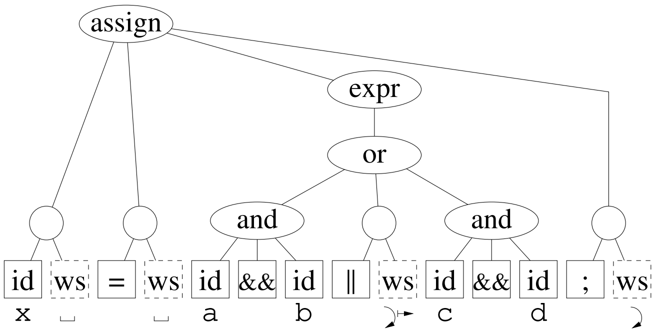


Figure C.4: The structural representation of the program text (*concrete syntax*) in Figure C.3.

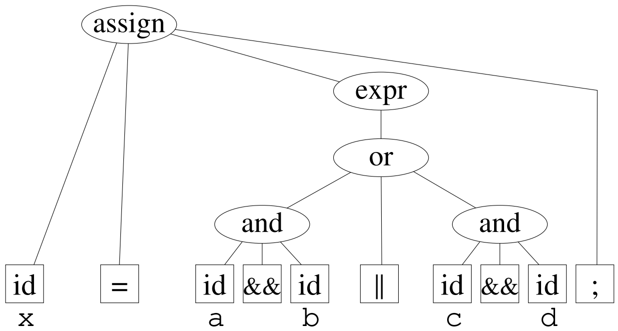


Figure C.5: View of the concrete syntax with whitespace removed. Additional elements of the concrete syntax, such as punctuation, could be removed by further abstracting the structure.

root →	bos assign eos	*Add sentinel root.*
assign →	id = expr ;	(*Original start symbol*)
expr →	expr && expr	
|	expr —— expr	
|	id	
. . .		
ws-pair →	WS WS	*Pair productions are shared*
|	WS ws-pair	*by all whitespace sequences.*
|	ws-pair WS	
|	ws-pair ws-pair	
id →	ID	*For each terminal symbol,*
|	id-conn	*allow a whitespace connector*
id-conn →	ID WS	*in its place.*
|	ID ws-pair	
and →	AND	
. . .		

Figure C.6: Whitespace support through grammar transformation. The grammar of Figure C.3 is transformed to allow each terminal symbol to be followed by an optional sequence of whitespace tokens. `WS` and `ID` represent whitespace and identifier tokens, respectively. The transformation applied to identifiers would be repeated for each terminal symbol (`equals`, `and`, etc.) and `bos`.

C.3 Construction Method I: Grammar Transformation

The representation described in the previous section can be constructed in various ways. In this section we describe a method based on *grammar transformation*, where the presence of whitespace is (usually) implicit in the source grammar supplied by the language description writer, but is made explicit in the transformed grammar used to generate the incremental analysis tools. In the following section, an alternate method is presented that instead modifies the incremental parsing algorithm and leaves the grammar unchanged.

The meta-language used to write grammars for language descriptions provided to the ISDE is typically an extended BNF; often regular expressions are permitted on the right-hand side of productions. In order to generate incremental analysis and transformation tools, the environment will translate the extended grammar to a canonical form. As part of this or a subsequent transformation, we can convert the *implicit* presence of whitespace to an *explicit* representation in the transformed grammar.[6] Figure C.6 provides a template for this expansion, illustrating how one terminal symbol from our running example is re-written to permit a sequence of whitespace tokens to follow it.

Each terminal symbol must be transformed in a similar fashion, introducing an accompanying connector production. In addition, `bos` must be similarly transformed and a production added to represent the root sentinel. (Since whitespace is attached to the preceding terminal symbol, `eos` does *not* require this transformation.) The four productions used to define `ws-pair` are included only once. These productions are written in a form that expresses the inherent associativity in the sequence, allowing a balancing algorithm to be employed. This notation will generate conflicts when the grammar is compiled by a deterministic parser generator such as `bison`. The default conflict resolution schemes ('prefer-shift', 'prefer-earlier-rule') can be safely used to produce a correct parser. (The same approach should be used when constructing parse tables for a GLR or other non-deterministic parser, to prevent the parser from constructing alternative interpretations of each whitespace sequence.)

Classes of grammars typically used for deterministic parsing (LR(k), LALR(k), and LL(k) for $k \geq 1$) are closed under this transformation, since the set of whitespace tokens is mutually exclusive with the grammar's original set of terminal symbols, and an entire whitespace sequence can be constructed using a single item of lookahead.

The grammar transformation approach is preferable when the compilation of language descriptions is easily extended. The environment *per se*, including the incremental parser, is unchanged. This approach is also preferable when it is necessary to describe languages in which whitespace can play a syntactic role, such as Haskell's 'off-side' rule [46]. In such languages, whitespace tokens will appear explicitly in the grammar; names for explicitly mentioned whitespace tokens must be integrated with names introduced by the transformation process.[7]

[6] This transformation is similar to that proposed by Visser [96], although no attempt is made there to produce an incremental evaluator from the result. The idea is also similar to the 'expected' comments in the Eiffel grammar [67].

[7] Even with the ability to name whitespace explicitly in the grammar, Haskell is non-trivial to describe in a manner suitable for incremental analysis. The lexical description must discover indentation changes and encode them in the set of whitespace tokens, and the language specification must accommodate the possibility of both explicit and implicit (whitespace-based) scope delimiters. The latter violates the assumption that the original set of terminal symbols

```
ACTION IncrementalParser::next_action (NODE *lookahead, int parseState) {
  Lookahead is either bos or a CN node that wraps bos. Shift into the parser's initial state.
  if (stack.is_empty()) return SHIFT 0;

  The top two nodes on the stack and the lookahead symbol are sufficient to 'parse' whitespace.
  NODE *tos = stack.element(0), *prev;
  int tosType = tos→type, prevType = BadType, laType = lookahead→type;
  if (stack.depth() > 1) {prev = stack.element(1); prevType = prev→type;}

  Previous/current item are one of <WS WS>, <WS PR>, <PR WS>, <PR PR>. Reduce them to a PR node.
  if ((tosType == WS || tosType == PR) && (prevType == WS || prevType == PR))
    return REDUCE PR_RULE;

  Create a wrapper when the whitespace sequence is complete.
  if (tosType == PR && laType != PR && laType != CN && laType != WS)
    return REDUCE CN_RULE;

  Treat an unmodified connector appearing as lookahead as its first child.
  if (laType == CN)
    return next_action(lookahead→first_child(), parseState);

  Backtrack from reductions to expose a terminal symbol when necessary to connect new adjacent whitespace tokens.
  if (is_nonterminal(tos) && tosType != PR && (laType == PR || laType == WS))
    right_breakdown();

  Now it's safe to shift PR and whitespace tokens; note that the parse state remains unchanged.
  if (laType == PR || laType == WS) return SHIFT parseState;

  The 'grammatical' case: look in the parse table to decide on the action.
  return parseTable→next_action(parseState, lookahead);
}
```

Figure C.7: Extensions to the incremental parser's `next_action` method. Existing whitespace subtrees are shifted onto the parse stack without changing the parse state. New whitespace material is 'parsed' to create a left-recursive chain that will be (re)balanced when parsing is complete. When a connector node appears as the parser's lookahead symbol, its first child is used to determine the next parse action. Connecting whitespace to the preceding terminal symbol may require that terminal symbol to be exposed by removing the right-hand edge of the enclosing subtree; this is accomplished by the `right_breakdown` routine, shown in Figure 6.2. '`WS`' is used as a shorthand to represent all whitespace token types, which are treated as an equivalence class by this algorithm. 'CN' and 'PR' denote connector and pair nodes, respectively (Figure C.1).

C.4 Construction Method II: Parser Modification

The grammar transformation described in the previous section is a simple mechanism for building the whitespace representation. In some circumstances, however, existing parse tables may need to be retained, or it may be appropriate to avoid additional transformations. Thus we present a second method for building the whitespace representation, based on directly modifying an incremental shift/reduce parser.

Rather than modify the grammar (and therefore the parse tables), we can simulate the additional shift and reduce actions by directly modifying the parsing algorithm instead—the `next_action` method is extended to handle the necessary operations as special cases. If no special rules apply, then this method will simply interrogate the parse table for the next action to take. The top two nodes on the parse stack and the lookahead symbol are sufficient to determine the next action in all cases; we do not change the parse state when shifting whitespace-related nodes. Note that this method, like the previous one, will shift a subtree that represents a whitespace sequence in constant time when it appears in the parser's input stream. Figure C.7 summarizes the necessary changes to the incremental parser's `next_action` routine.

The parser-based approach can be implemented directly in both sentential-form and state-matching incremental parsers. A similar technique can be used to extend an incremental GLR parser (Chapter 7) by encoding a synthetic 'whitespace state' in the nodes of its graph-structured stack.

---

is mutually exclusive with the set of whitespace tokens. Incremental GLR parsing (Chapter 7) can be used to overcome any conflicts induced by relaxing this restriction, using non-deterministic parsing to try several possibilities simultaneously.

When parsing is complete (the parse table indicates an *accept* action), the parse stack will contain two elements. The first is either the `bos` token itself or a connector node whose left child is `bos`. The second parse stack entry will be a node corresponding to the start symbol of the grammar. At this point the lookahead symbol, `eos`, can be pushed onto the stack and the three stack elements 'reduced' to create the sentinel root node of the program representation (Figure 3.2).

Error *detection* is essentially unchanged by the presence of whitespace; any whitespace tokens immediately preceding the point of detection will typically be gathered into their sequence representation before the error is discovered. The impact of whitespace material on error *recovery* is discussed in Chapter 8.

The representation constructed by modifying the incremental parser is structurally equivalent to that produced by grammar transformation up to the balancing of sequences; the only difference is that a *single* connector type can be used in this approach, since there is no need to ensure closure properties required by the transformation-based scheme.

Colophon

This document was prepared on a Unix™ system with the LaTeX formatting system using the `book` style. Indentation following chapter and section headings has been suppressed.

The bottom margin is ragged, producing a better vertical layout in general. TeX's line-breaking was set to sloppy and complaints of less than 5000 badness were ignored. PostScript™ images are included using the `epsf` packages. Bibliography management was done using John Boyland's `makebib` utility to merge bibliography databases, followed by BibTeX.

The main text is set in Times font, with margins adjusted to match the conventional binding style used for UCB technical reports. Program text is set in courier; algorithms in figures are typeset using John Boyland's `program` style. Most symbols were produced using TeX's math symbol facilities; however, '~' and 'C++' required explicit construction to achieve a pleasing visual style.

Tables were set directly in LaTeX. Graphics were produced using `xfig`, with the exception of Figure 7.4, which was produced by SuperMongo™ directly from the dataset. `dvi`-to-PostScript conversion was done with `dvipsk 5.58` from Radical Eye Software.™ The online version was created using `latex2html` on the technical report version of this document.