The *Pan* Language-Based Editing System

ROBERT A. BALLANCE
University of New Mexico
and
SUSAN L. GRAHAM and MICHAEL L. VAN DE VANTER
University of California

Powerful editing systems for developing complex software documents are difficult to engineer. Besides requiring efficient incremental algorithms and complex data structures, such editors must accommodate flexible editing styles, provide a consistent, coherent, and powerful user interface, support individual variations and projectwide configurations, maintain a sharable database of information concerning the documents being edited, and integrate smoothly with the other tools in the environment. *Pan* is a language-based editing and browsing system that exhibits these characteristics. This paper surveys the design and engineering of *Pan*, paying particular attention to a number of issues that pervade the system: incremental checking and analysis, information retention in the presence of change, tolerance for errors and anomalies, and extension facilities.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques; D.2.3 [**Software Engineering**]: Coding—*program editors*; D.2.6 [**Software Engineering**]: Programming Environments; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces

General Terms: Design, Documentation, Languages

Additional Key Words and Phrases: Coherent user interfaces, colander, contextual constraint, extension facilities, grammatical abstraction, incremental checking and analysis algorithms, interactive programming environment, Ladle, logic programming, logical constraint grammar, Pan, reason maintenance, syntax-recognizing editor, tolerance for errors and anomalies

1. INTRODUCTION

Languages and documents play a significant role in software development. In addition to the natural language used for written human communication, developers use a variety of more formal languages to describe both software

This research was sponsored in part by the Defense Advanced Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command under contracts N00039-84-C-0089 and N00039-88-C-0292, by IBM under IBM research contract 564516, by a gift from Apple Computer, Inc., and by the State of California MICRO Fellowship Program. A preliminary version of this paper was presented at the Fourth ACM SIGSOFT Symposium on Software Development Environments, December 1990.
Authors' addresses: R. A. Ballance, Department of Computer Science, University of New Mexico, Albuquerque, NM 87131; S. L. Graham and M. L. Van De Vanter, Computer Science Division (EECS), University of California, Berkeley, CA 94720.

products and the processes by which they are developed and maintained. Some examples are design languages, specification languages, structured-documentation languages, programming languages, and numerous small languages for scripts, schemata, and mail messages. Furthermore, programs often contain embedded "little languages" that impose their own conventions. For example, many subroutine libraries define minilanguages for long and complex argument sequences.[1]

The *Pan*[2] editing and browsing system originated from an investigation into ways to exploit language-based technology to provide more integrated support for software developers and for the documents with which they work. The design of *Pan* rests on the premise that the bridge between developers and their software will be intelligent editing interfaces, namely, interfaces that provide a generalization of the services of a traditional interactive editor to support interactive browsing, manipulation, and modification of documents.

The current implementation of *Pan* [9, 21] is a fully functional prototype. It supports ongoing research in language description, language-based analysis techniques, user-interface design, advanced program-viewing methods, and related areas. The functional characteristics of this prototype were chosen for maximum leverage as a usable tool and as a platform for continuing research.

—*Pan* is a multiwindow, multiple-font, mouse-based editing system that is fully customizable and extensible in the spirit of Emacs [57].
—*Pan* incrementally builds and maintains a collection of information about documents that can be shared with other tools.
—*Pan* users can freely mix text- and language-oriented manipulations in the same visual editing field; text editing is completely unrestricted.
—A single *Pan* session may involve multiple languages.
—New languages can be added to *Pan* by writing language descriptions. Extension mechanisms for other language services are also provided.

New language description techniques were developed for *Pan*. *Grammatical abstraction* establishes formal correspondence between the concrete (parsing) syntax and the abstract syntax for each language [8, 12]. *Logical constraint grammars* are an adaptation of logic programming and consistency maintenance for the specification and enforcement of contextual constraints [6, 7]. Information gathered during constraint enforcement is retained in a memory-resident logic database (available to other tools) and revised incrementally as documents change.

Related issues, including support for novice programmers and learning environments, support for program execution, graphical display and editing, and the design and implementation of a persistent database were deferred.

[1] Libraries for window systems often have these kinds of interfaces.
[2] Why "Pan"? In the Greek pantheon, Pan is the god of trees and forests Also, the prefix "pan-" connotes "applying to all," in this instance referring to the multilingual text- and structure-oriented approach adopted for this system Finally, since an editor is one of the most frequently used tools in a programmer's toolbox, the allusion to the lowly, ubiquitous kitchen utensil is apt.

This paper reviews the goals and early design decisions for *Pan* and surveys the implementation of the *Pan* prototype. The discussion emphasizes the interactions of the technologies and components, and, in particular, how seemingly simple design strategies pervade the system. Detailed discussions of *Pan*'s components and underlying technology have been presented elsewhere [7, 8, 66].

Throughout the paper the term *language-based* indicates that one or more of the facilities provided by the system make use of language-specific information derived from the documents known to the system. In the context of this paper, the term *system* (or *editing/browsing system*) encompasses the entire collection of services that are used to browse, manipulate, and modify one or more documents interactively.

2. LANGUAGE-BASED ENVIRONMENTS

The *Pan* project was motivated by a vision of language-based browsing and editing systems as the primary interface between people and integrated environments containing the documents they manage. Positioned this way, between users, tools, and documents (Figure 1), such a system is uniquely situated to gather and present information *about* documents for the benefit of its users.

2.1 The Needs of Developers

The developers who read and write software-related documents are experienced professionals. They are proficient with their primary tools and languages and are usually skilled at authoring documents in those languages. Like all users, they may need extra support when confronted with unfamiliar languages or documents.

As Winograd [68] and Goldberg [25] argue, software systems have become so large and complex that developers spend far more time reading, understanding, modifying, and adapting documents than they do creating them in the first place. These activities involve a variety of subtasks, in particular, the acquisition and exploitation of many kinds of information [29, 41]. A successful interactive development environment must recognize, gather, and present complex information about documents suitable for the particular task at hand. In order to provide such services, the system must maintain a model of the syntactic, static semantic, and contextual properties of the document. At the same time, there must be minimal disruption of services in midtask, when documents may be incomplete and inconsistent.

An editing interface must not compromise flexibility and power, even for safety or learnability. For example, many language-based editors restrict how and when users may manipulate documents as *text*, in order to maintain syntactic correctness. But both contrary arguments [67, 69] and experience suggest that experienced developers generally will not tolerate restrictions on a natural and convenient mode of text-based interaction. Likewise, a system must not "do too much" [46] in the form of gratuitous intrusive services.

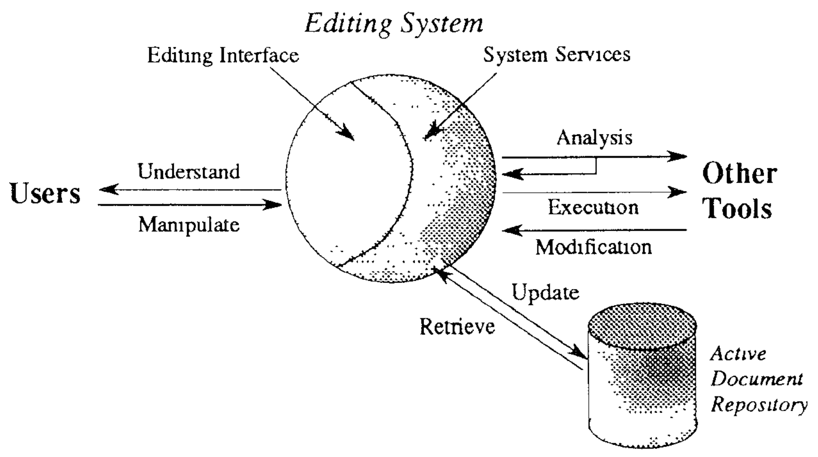


Fig. 1 Editing interface and system services in relation to the environment.

Finally, the way developers work changes over time; projects and conventions (community-wide, personal, and project-specific) come and go. Even languages themselves change. Old languages are revised and extended; new languages are adopted and sometimes invented for specific needs. Development environments and their services must evolve or be abandoned.

2.2 The Nature of Software Documents

The documents managed by software developers constitute richly connected, overlapping webs of information having many structural, as well as textual, aspects. Some structure is syntactic, for example, paragraphs in reference manuals, data definitions in design documents, and statements in programming languages. Other structure stems from the content of the documents.

Some document structure can be derived automatically from knowledge of underlying formal languages, for example, the connection between a figure and references to it in a book, the relationships between declarations, definitions, and uses of variables in a computer program, the call graph of a program, and the relationships among grammatical units defined by a formal syntax. In other cases, important structure is independent of language, for instance, hyperlinks in a hypertext system and the user-imposed hierarchy supported by outline processors. Such structure cannot be inferred and must be retained when provided by developers. The editors ED3 [60] and Tioga [63] and the noninteractive WEB [38] are examples of systems that support user-supplied structure.

Some documents are encoded in more than one language, in which case important relationships may cross language boundaries. The Mentor programming environment [20] supports such nesting of languages.

Although formal syntax is often presumed to be the most useful structure for the purpose of document display and user interaction, other kinds of structure are at least as important. For example, language-based formatting (prettyprinting), which is intended to aid user understanding of a document, traditionally is based only on surface syntax. Formatting is considerably more helpful if it is sensitive to scopes, types, and def-use relationships, as well as to local conventions and even to distinctions such as "mainline" versus "error-handling" code. Different users and different user tasks require different uses of structure and different forms of access to the information within documents. Although the information must be broad in subject domain, it need not be deep (in the sense that program *plans* [42, 56] and *clichés* [54] are deep) to be useful.

2.3 The Benefits of Integration

Complex, expensive analyses in such a system are economical only when many tools share the resulting information. The computation that verifies a document's type correctness can also provide information useful to a compiler, a global interface checker, or an auditing tool. Conversely, information produced by other tools should also be made visible through an editing interface. For example, helpful views of programs might exploit measurement results and version history. In addition, as the formatting example suggests, system services are enriched if user-supplied information is preserved and used widely.

3. DESIGN OF THE *Pan* SYSTEM

In this section we summarize the major design strategies for *Pan*, the system organization, the text editing interface, and those extension and customization facilities other than language description. Language-based technology, mechanisms, and services are described in Sections 4–6.

3.1 Design Strategies

Our vision of the role to be played by *Pan* led to the adoption of a small number of pervasive and surprisingly interdependent strategies for its design.

Text-based interface. The text editing services and user interfaces of *Pan* are designed around familiar models, to encourage smooth integration into existing working environments. High-quality typography is used for the visual presentation of text, a feature often not realized in editing interfaces. Studies by Baecker and Marcus [4] and by Oman and Cook [48] suggest the value of typography for program presentation.

Language description. *Pan* supports many languages, driven by a *description* of each. The descriptive medium is largely declarative. It supports the *definition* of languages, but also includes specifications for usage conventions, user interaction, and language-specific services.

Syntax recognition. To present the appearance of a "smart text editor," one that also supports language-based interaction, *Pan* is *syntax-recognizing*, as are Babel [31], the Saga editor [37], and SRE [11]. A syntax-recognizing system is one in which the user provides text and the system infers the syntactic structure by analysis. In contrast, we call systems like the Cornell

Program Synthesizer [61], Mentor [20], and Gandalf [26] *syntax-directed* since the primary mode of editing in those systems is template based. The syntax-recognizing approach does not preclude a user interface that simulates syntax-directed editing. A simple prototype has convinced us that syntax-directed editing can be provided easily in a syntax-recognizing editor. Similarly, most syntax-directed editing systems support some localized text editing. As a by-product of syntax recognition, *all* language-oriented information, including the primary internal tree representation shared with other analyses, is derived originally from a textual representation.

Incrementality. Maintaining full service during editing demands that derived information be revised as documents change. *Pan* is organized around update algorithms rather than recomputation. This approach enhances the retention of previously determined information (some of which may be user-supplied or otherwise nonrecoverable) and maintains computational efficiency in the face of collective document growth. Users are seldom willing to compromise on speed, even for enhanced functionality.

Tolerance for variance. During the unrestricted text-oriented editing permitted by syntax recognition, documents are most often ill-formed with respect to the underlying language definition. Maintaining full service demands that no more restrictions be placed on the user in this situation than a standard text editor does in the presence of spelling errors. To emphasize the distinction between this approach and those adopted by many language-based editors, we refer to *variances* rather than to the traditional term *language errors*. This approach acknowledges that experienced users often introduce variances deliberately while working toward a desired result; users should not be penalized by the system's failure to understand the process [43].

Coherent user interface. The shift of emphasis from the preemptive "language error" to the informative "variance" is only one example of ways in which the details of language-based technology and implementation should be concealed. Following the view that *Pan* is an interface between user and document, language-oriented interaction is organized around a conceptual model of document structure, tuned for each language to be convenient and natural. Users are offered a variety of *services* that exploit rich internal data while hiding representational complexity.

Extensibility and customization. *Pan* is designed for convenient adaptation to variations among users, projects (group behavior), and sites. As a research platform, it accommodates extension and evolution [40]. Language description is only one aspect of *Pan*'s extensibility.

3.2 System Organization

Pan's implementation supports users in three categories: clients, customizers, and language description authors.[3] The classes of users can be characterized

[3] It is the role of the user that is the issue here. All but the most naive client users customize their environments in simple ways, blurring the boundary between clients and customızers

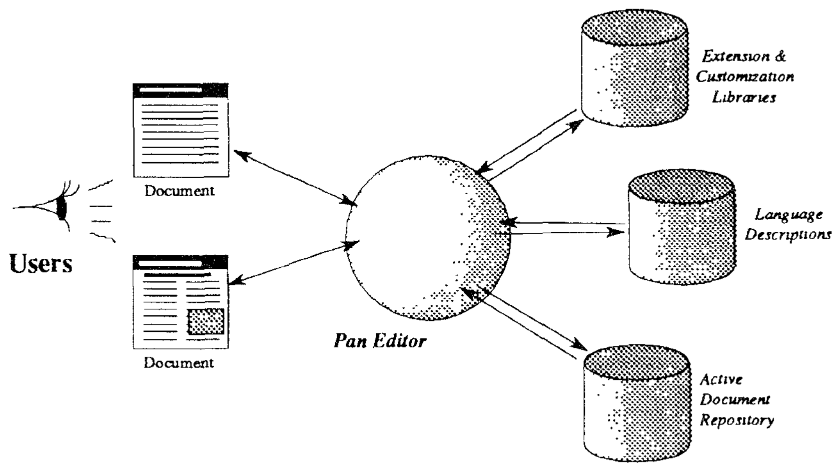

Users
Document
Document
Pan Editor
Extension & Customization Libraries
Language Descriptions
Active Document Repository

Fig. 2. System oganization: Users' perspective.

by the tasks they perform and by the knowledge they need about various aspects of the system (Figure 2).

For *clients*, *Pan* is the interface for browsing and editing documents. Section 3.3 describes the surface characteristics of this interface, in particular, the rich text editing and presentation facilities available in every context. Clients use libraries and language descriptions but need to know little about them.

Customizers, on the other hand, augment extension and customization libraries; this task requires expertise ranging from the shallow (e.g., adding key bindings) to the deep (e.g., adding a new kind of directory editor). Section 3.4 describes some of the mechanisms that support customization.

Language description authors must be familiar with language processing and with the techniques used by *Pan*'s document analyzer, described in Sections 4 and 5. A language description contains information about syntactic structure and context-sensitive constraints. Constraints usually include the static semantic rules of the language, but can also include site-specific or project-specific restrictions such as naming conventions.

As part of the emphasis on *coherent user interfaces* [66], language descriptions also specify user interaction with generic language-based services (described in Section 6) and may add new language-specific services. Multiple descriptions for a single underlying language may coexist, each providing a different interface for a different class of users. In this broader view, the author is a user-interface designer and the language description a user-interface specification.

3.3 Text-Based Editing Interface

Superficially, *Pan* appears to clients as a convenient bit-mapped, mouse-based, multiple window text editor in the spirit of Bravo [39] and its many successors. Figure 6 of Section 6 shows a sample editing session. But *Pan* is also an editor that happens to be extremely knowledgeable about document structure and the local working environment: languages in use, local conventions, and perhaps the user's own personal working habits.

The same text-oriented services are provided for every document, whether or not it is written in a language that *Pan* is prepared (by prior language description) to analyze. Users familiar with Emacs [57] find the transition between the two editors smoothed by compatible key bindings [21] and comparable text services. Generalized undo, kill-rings, text-filling, customization, extension, and self-documentation are among *Pan*'s standard services.

In contrast to many syntax-directed editors, one might use *Pan* for editing text without ever giving a thought to its other capabilities. But at any time one may choose to broaden the dialogue with *Pan* and to exploit information (maintained by *Pan*) *about* the document. *Pan* can be directed to use this information to guide editing actions, to configure and selectively to highlight the textual display, to present answers to queries, and more. Some of these services are described in more detail in Section 6.

Two general (and configurable) mechanisms enable the persistent display of information about documents: flags and visual text attributes. A *flag* is a small glyph near the top right of a window; it may appear and disappear, change shape, change color, or all three in response to a change of state concerning the document. This document state might be whether it has been modified, whether it may be modified, whether particular services are enabled, whether certain kinds of inconsistency have been detected, or any other property that can be represented as an editor variable (Section 3.4).

A *visual text attribute* may be applied independently to any group of characters in the display by *Pan*'s services: choice of font (from a configurable *map* for each document, specifying up to eight fonts, proportionally spaced and varying in height), choice of ink color, and choice of background color for highlighting. The user's current text selection (shared by all windows on a document) is underlined, independent of any other attributes.

3.4 Customization and Extension

Pan's services are built upon a rich and flexible base, designed for experimentation with document analysis techniques and the design of editing interfaces. The language-based mechanisms described in this paper use the infrastructure, as do a few experimental services that are not yet language-based: a browsing interface to the file system, a hypertext-like browser for UNIX[4] man pages, and an elaborate internal help and documentation system that can be configured for each category of user.

Although some of *Pan*'s customization and extension facilities are declarative, others require programming. Unlike Emacs, we chose to provide access to *Pan*'s implementation language (Common LISP) and its run-time system, rather than inventing a separate extension language. Although that access can be abused, it has not proved to be a problem in practice. *Pan*'s extension

[4] UNIX is a trademark of AT&T Bell Laboratories.

language is embedded in Common LISP and is supported by a number of general mechanisms, all integrated with the on-line help system. Many of *Pan*'s current services are implemented in the extension language.

Variables. In *Pan, editor variables* are scoped dynamically, allowing bindings per document instance, per document type, and globally. This form of scoping permits the organization of services at each level, including language-specific modes. Many fundamental mechanisms are built using variables, including user options, keyboard bindings, menus, character classes, flags, font maps, color maps, hooks, and window configuration, all of which respect scoping at a relatively fine granularity. New variables may be defined at any time using a declarative syntax that supports options such as predicates for type checking, notifiers for active values, and restrictions on the scopes in which the variable may be bound.

Function and macro definition. Functions and macros may be defined at the extension level, implying automatic integration with *Pan*'s internal documentation system and its undo system. New functions and macros may be defined at any time during an interactive session, along with declaratively specified options. Those functions (called *commands*) that are bindable to keystrokes and menus are automatically integrated into the run-time command dispatch mechanism. Command arguments are specified with a declarative syntax that directs the command dispatcher to collect values dynamically (by user selection, by configurable prompters, or by default) with automatic type checking and error recovery.

Generic exception handling. Pan distinguishes three categories of exceptions that may be signaled (along with a message) without concern for context: announcement, warning, and error. Response to *Pan* exceptions is context-sensitive, implemented by dynamically bound exception handlers [64]. For example, an error might arise while a language description is being loaded. Language descriptions can be loaded in any of three contexts. First, descriptions may be preloaded when a new *Pan* is being built. In this case, the error terminates the build with a logged message. Second, descriptions may be automatically loaded during a session with *Pan*. The handler in this case displays the message, beeps, and resets the command dispatcher, effectively terminating the command that triggered language loading. Third, descriptions may be loaded directly from within a Common LISP debugging loop. In this case, a recursive call to the debugger preserves the stack for inspection. Extension-level primitives guard their internal state and respect conventions for signaling exceptions; simple extension code can ignore exception handling and still be robust.

4. DOCUMENT ANALYSIS

Document analysis in *Pan* relies on two components: *Ladle* (Language Description Language) [12] and *Colander* (Constraint Language and Interpreter) [6]. *Ladle* manages incremental lexical and syntactic analysis; it includes both an off-line preprocessor that generates language-specific tables and a run-time analyzer that revises *Pan*'s internal document representation to reflect textual changes. *Colander* manages the specification and incremental checking of contextual constraints. Like *Ladle*, *Colander* includes both an off-line preprocessor and a run-time component. The editing interface (Section 6) coordinates analysis and makes derived information accessible to users and client programs.

This section describes each of *Ladle* and *Colander* in a bit more detail, discusses how the two cooperate, and finally examines some important design issues that cross all component boundaries.

4.1 Language Description Processing

A *Pan language description* contains declarative information for use by each of *Pan*'s three components:[5]

(1) Lexical and syntactic data, used by *Ladle*, describe the syntax of the language and define an internal tree-structured representation (Section 4.2).
(2) The *Colander* portion specifies context-sensitive constraints, including, but not limited to, the static semantics of the language (Sections 4.3 and 5). This specification may also direct that certain data derived during contextual-constraint checking be stored and made available for general use.
(3) User-interface specifications configure the editing interface for the language (Sections 6.1 and 6.2).

Figure 3 illustrates the flow of information from a language description to the run-time *Pan* system, for either preloading or dynamic loading at run time.

Multiple *Pan* language descriptions could be written for a single language, suited for different users and different tasks. The primary motive would be to provide different collections of services of the kind described in Section 6. However, one could also experiment with different presentation styles, alternate internal representations (abstract syntaxes), or different styles of *Colander* description. One area of active research concerns the layering of language descriptions so that multiple views of a single abstract syntax can be managed effectively.

A related area of research concerns sharing structures and description components among language descriptions. Many languages share similar semantic, as well as syntactic, concepts. The reuse of portions of the language description simplifies the language description writer's task and makes it possible to share data and establish linkages among documents written in different languages.

[5] In the current implementation, each document must be composed using a single language Our architecture and algorithms support documents composed from multiple languages, but the current implementation does not.

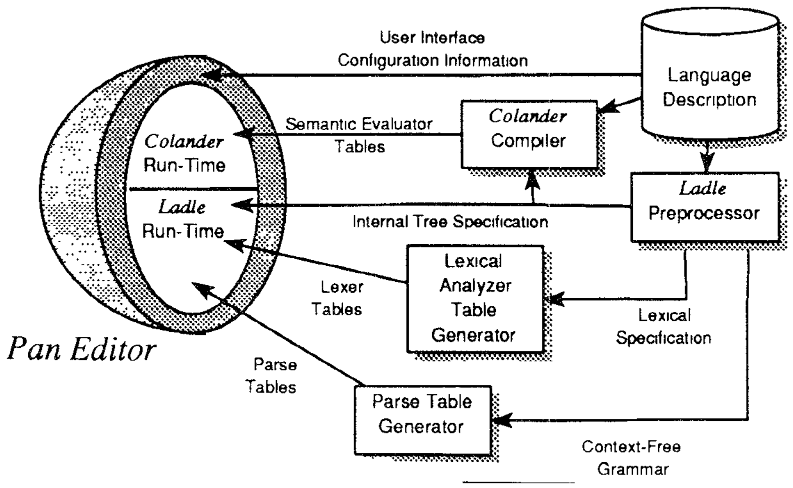


Fig. 3. Language description processing.

4.2 *Ladle*

An abstract syntax is described to *Ladle* using an augmented context-free grammar, which also specifies the tree-structured representation. By defining the semantically relevant structures of the language, the grammar implicitly defines the terms in which the rest of *Pan* accesses and manipulates document components.

When text-oriented editing of syntactic structures is to be supported, additional information enables *Ladle* to convert textual representations to tree-structured representations and vice versa:

—The lexical description may include both regular expressions and bracketed regular expressions, that is, expressions with paired delimiters such as quote marks. Bracketing can be either nested or simple.

—The grammar for the abstract syntax is augmented by specifying those productions necessary to disambiguate the original (abstract) grammar or to incorporate additional keywords and punctuation. *Ladle* constructs a full parsing grammar from the additional productions and the grammar for the abstract syntax.

—Optional directives tune the *Ladle* syntactic error-recovery mechanisms invoked during parsing. These directives also have important effects on the editing interface (Section 6.1).

Internally, *Ladle* manipulates two context-free grammars: one describing the abstract syntax and the other used to construct parse tables. The two must be related by *grammatical abstraction*[6] [8], a relation ensuring the

[6] More recently, Butcher has recast this work in terms of *grammatical expansion* [12].

$$\begin{array}{lllll}
1) & \langle \mathit{stmt} \rangle & \rightarrow & \langle \mathit{if\text{-}stmt} \rangle \text{ ";"} \\
2) & \langle \mathit{if\text{-}stmt} \rangle & \rightarrow & \langle \mathit{if\text{-}part} \rangle \langle \mathit{else\text{-}part} \rangle \\
3) & \langle \mathit{if\text{-}part} \rangle & \rightarrow & \mathsf{if}\ \langle \mathit{expr} \rangle\ \mathsf{then}\ \langle \mathit{stmts} \rangle \\
4) & \langle \mathit{else\text{-}part} \rangle & \rightarrow & \epsilon \\
5) & & \mid & \mathsf{else}\ \langle \mathit{stmts} \rangle
\end{array}$$

Concrete Grammar G_1

$$\begin{array}{llll}
1') & \langle \mathit{stmt} \rangle & \rightarrow & \mathsf{if}\ \langle \mathit{expr} \rangle\ \mathsf{then}\ \langle \mathit{stmts} \rangle \text{ ";"} \\
2') & & \mid & \mathsf{if}\ \langle \mathit{expr} \rangle\ \mathsf{then}\ \langle \mathit{stmts} \rangle\ \mathsf{else}\ \langle \mathit{stmts} \rangle \text{ ";"}
\end{array}$$

Abstract Grammar $\widehat{G_1}$

$$\begin{array}{llll}
1'') & \langle \mathit{stmt} \rangle & \rightarrow & \langle \mathit{expr} \rangle \langle \mathit{stmts} \rangle \\
2'') & & \mid & \langle \mathit{expr} \rangle \langle \mathit{stmts} \rangle \langle \mathit{stmts} \rangle
\end{array}$$

Abstract Grammar $\widehat{G'_1}$

Fig. 4. Grammatical abstraction.

following:

(1) The abstract syntax represents a less complex version of the concrete syntax, but structures of the abstract syntax correspond to structures of the concrete syntax in a well-defined way. Singleton derivation steps and nonterminals needed primarily for parsing can be suppressed. Keywords and tokens of the concrete syntax that can be inferred from the abstract structure need not be explicit in the abstract syntax.

(2) Efficient incremental transformations from concrete to abstract and from abstract to concrete can be generated automatically; no action routines or special procedures are necessary. The transformation from concrete to abstract is triggered directly by actions of the parser.

(3) The transformation from concrete to abstract is reversible, so that relevant information about a concrete derivation can be recovered from its abstract representation. This property allows the system to parse modifications to documents incrementally without having to maintain the entire parse tree.

(4) The relationship between the two descriptions is declarative and statically verifiable so that developers can modify either syntax description independently. This approach allows a high degree of control over both the structure of an internal representation and the behavior of the system during parsing.

Grammatical abstraction is structural; it does not use semantic information to identify corresponding structures. Two examples of grammatical abstraction appear in Figure 4. The concrete grammar G_1 describes the syntax of conditional statements. The fragments $\widehat{G_1}$ and $\widehat{G'_1}$ are both allowable (but different) grammatical abstractions from the fragment G_1.

The *Ladle* preprocessor generates the tables needed to describe the internal tree representation as well as auxiliary tables needed during incremental parsing and error recovery. A standard lexical analyzer generator and a modified LALR(1) parser generator are also invoked, as shown in Figure 3.

To date, syntactic descriptions have been written for Modula-2, for Pascal, for Ada, for *Colander*, and for *Ladle*'s own language description language. Descriptions are being developed for a variety of other languages, including C, C++, and FIDIL [28].

4.3 Colander

Colander supports the description and incremental checking of contextual constraints. Constraints include nonstructural aspects of a language definition such as name binding rules and type consistency rules, as well as extralingual structure. Examples of the latter include site- or project-specific naming conventions, design constraints, and complex, nonlocal linkages within or among documents.

Our approach is based on the notion of *logical constraint grammars*. In a logical constraint grammar, a context-free grammar is used as a base. Contextual constraints are expressed by annotating productions in the base grammar with goals written in a logic programming language.[7] An incremental evaluator monitors changes to the document and the derived information in order to maintain consistency between them.

To date, logical constraint grammars have been used to define the static semantics of programming languages, including Modula-2, to express some aspects of design semantics, and to describe and maintain prettyprinting information. Other problems that can be expressed using logical and constraint grammars include the kinds of analyses performed by tools such as Masterscope [44] or Microscope [2].

Colander itself has three subcomponents: a compiler, a consistency manager, and an evaluator. The *Colander* compiler generates the code used by the evaluator[8] as well as the run-time tables required for consistency maintenance. The consistency manager, a simple reason maintenance system [22, 55], invokes the evaluator to (re)attempt a goal. The evaluator, in turn, collects the information maintained by the consistency manager. Section 5 presents *Colander* in more detail.

4.4 Document Processing

Textual changes are incorporated into an internal tree in two phases: lexical and parsing. *Ladle*'s incremental lexical analyzer synchronizes a stream of lexemes with an underlying text stream, updating only the changed portions of the lexical stream. The lexical analyzer maintains a summary of changes for use by the incremental parser.

[7] Logical constraint grammars should not be confused with *constraint logic programming* [15], a generalization of logic programming.

[8] The compiler uses *Pan* to parse language descriptions. This involution is one example of how *Pan* is used to support itself.

Ladle's incremental LALR(1) parser revises the tree-structured representation in response to lexical changes. This parser can create a tree from scratch, but in response to lexical changes, it need only modify affected areas of the tree. It uses a variant of an algorithm by Jalili and Gallier [33]. During incremental parsing, the algorithm first "unzips" the internal tree along a path between the root and the leftmost changed area. The algorithm concludes by incorporating changes and "zipping up" the unzipped portion. When unzipping and zipping up, subtrees are broken apart and then reconstituted.

For the benefit of *Colander* and other clients, *Ladle* classifies tree nodes after each parse: newly created, deleted from tree, reconstituted, and unchanged. Semantic values associated with reconstituted nodes are retained, even though their annotations may require updating.

Pan's distinction between syntax and contextual constraints (or static semantics) reflects a division common to almost all language description techniques. It creates problems in practice for languages in which parsing and semantic analysis must be intertwined [23], for example, the well-known "typedef" problem in the "C" language. Research into general solutions to these problems within *Pan* is currently under way.

4.5 Information Retention

Since subtrees may be heavily annotated (both by tools and by users), actual changes to the internal tree must be minimized to avoid needless information loss. A simplistic implementation of the incremental parsing algorithm would destroy and then recreate every subtree between a changed area and the tree root. Widely shared data often appear close to the root of the tree. The loss of semantic annotations on those nodes would cost lengthy and often unnecessary recomputation, so efficient incremental constraint checking by *Colander* depends on *Ladle*'s reuse (either physical or virtual) of these nodes.

Ladle uses an effective heuristic for reconstituting unzipped nodes. The parser keeps a stack of "divided" tree nodes. When new nodes are needed, they are taken from this stack if possible. A node is reused when it represents the same production in the abstract syntax as in its previous use and when its leftmost child is unchanged between parses. The heuristic can err either by not reusing a node or by reusing a node in a different (absolute) position within the syntax tree. Neither case causes major difficulties, although either may entail extra computation during incremental constraint checking. In practice, the heuristic for reuse succeeds in the most critical cases, preserving portions of the internal tree close to the root.

4.6 Tolerance, Recovery, and Variances

Documents are most often incomplete and ill-formed during editing sessions. To maintain full service to the user, *Pan*'s analysis mechanisms treat such problems as *variances*, not as errors, and make every effort to treat them as interesting but relatively normal occurrences. *Pan*'s internal document representations are automatically extended to admit variances and to retain as much information as possible in their presence. Sections 6.1 and 6.2 describe how the editing interface makes available information about variances.

Lexical analysis normally succeeds, since error tokens are generated whenever there are unexpected characters, leading to parser-based syntactic variances. If lexical analysis fails, a lexical variance is signaled. For instance, an unterminated comment may lead to a lexical variance. A variance detected during lexical analysis inhibits both parsing and contextual-constraint checking, and all information that existed prior to the attempt to reanalyze the document is preserved.

During parsing, *Pan* uses a simple, effective, panic mode mechanism [17] for syntactic error recovery. The structures to which it recovers are those of the abstract syntax. Directives in the *Ladle* portion of language descriptions tune the recovery mechanism for each language. The presence of a syntactic variance is marked in the internal tree by an *error subtree* annotated with an appropriate error message. The children of an error subtree are the lexemes and subtrees that were skipped over during the recovery. This recovery strategy is similar to that used in the Saga editor [37]. Any extant annotations on the subtrees within the error subtree are preserved, including annotations created by *Colander*. Contextual constraints within an error subtree are not attempted by *Colander*. When the user corrects the variance, prior annotations can immediately be reused. This is just a special case of the general information retention problem.

Contextual-constraint checking can proceed in the presence of syntactic variances; any constraints within error subtrees are simply ignored. Unsatisfied contextual constraints form another kind of variance, resulting in annotations on offending nodes.

5. LOGICAL CONSTRAINT GRAMMARS

A great deal of *Pan*'s analytical power, as well as its potential for future extension, derives from the adaptation of logic programming and consistency maintenance, as introduced in Section 4.3. This section reviews the theoretical foundations of logical constraint grammars [6, 8] and describes *Pan*'s particular language and implementation, *Colander*, in more detail.

Logic programming is a natural paradigm for the specification, checking, and maintenance of contextual constraints. First, context-sensitive aspects of formal languages are often described informally using natural language that approximates logical structure. Translation of these descriptions into clausal logic is relatively straightforward. Second, the act of checking contextual constraints an be viewed as satisfying the constraints relative to some collection of information. (In pass-oriented applications like compilers, this collection of information is represented by a symbol table.) Finally, the presence of a logic programming language implies the existence of both an inference engine and a logic database. The extensibility of *Pan*, beyond conventional constraint checking, derives from the presence of the database as a shared repository and from the generality of the logic-based inference engine.

A logical constraint grammar (LCG) is a context-free grammar G in which symbols and productions have been annotated with goals, expressed in a logic-programming language, that specify constraints on the language generated by G. Goals are satisfied using backtracking search based on unification, as in PROLOG. An *evaluator* for an LCG description begins by executing the goals that are independent of any syntactic structure. These goals initialize the data shared by all of the documents written in a given language. The evaluator then attempts to satisfy all goals associated with syntactic structures present in the document. The evaluator stops processing whenever all of the goals are successfully proved or no further goals can be proved. A document is considered well formed whenever all of its associated goals have succeeded.

As in unrestricted attribute grammars, circularities among the goals in an LCG description can arise. It is left to the writer of an LCG description to remedy circularities.

Other well-known specification formalisms for language implementation include attribute grammars [18, 53], action routines [36, 45], context relations [5], and natural semantics [34]. Attribute grammars provide a declarative mechanism for defining attribute (property) values at subtrees in a syntax tree. An attribute value at a subtree is a function of the attribute values defined at neighboring subtrees; they may also depend on other values defined at the current subtree. Unfortunately, an attribute grammar itself usually specifies only the attributes and their interrelationships. A separate formalism, usually a general-purpose programming language, must be used to specify the semantic functions and the data types. Yet much of the interesting information in an attribute-grammar-based specification resides in the code that implements the semantic functions and data types. Using a separate formalism hinders the analysis of interactions within a language description. Another drawback to attribute grammars is that there is no easy way to make information in the attribute values available to other tools [32]. Finally, most algorithms for incremental evaluation of attribute grammars unduly constrain the kind of manipulations that an interactive system can support [66].

Action routines are arbitrary procedures associated with nodes in a syntax tree. An action routine is invoked whenever a particular event concerning its associated node occurs. The actual events that can cause action routines to be invoked depend on the implementation. Thus, action routines form a simple and powerful mechanism for implementing contextual constraint checking, but they require a developer to deal with all aspects of incrementality explicitly.

Context relations, as used in the PSG system [5], provide a novel means for composing fragments of syntax trees. Each program fragment is summarized by its context relation. A fragment is considered valid if it can be embedded into a correct (larger) fragment. A context relation summarizes the set of all "still-possible attribute values," the values that have not been ruled out by the evaluation of other context constraints. An empty context relation indicates that there is no possible valid assignment of values to the attributes of the fragment, and therefore indicates an error. The paper by Bahlke and Snelting [5] describes their incremental analysis algorithm. Context relations, like attribute grammars, relegate many of their description details, such as name-resolution rules, to a separate formalism.

Natural semantics [34] is a logic-based formalism based on Plotkin's structured operational semantics [49]. In natural semantics, a language description specifies an abstract syntax together with axioms and inference rules that characterize the structures in the abstract syntax. The collection of axioms and inference rules is identified with a logic based on natural deduction [50]. Reasoning about the target language is reduced to proving theorems in that logic. Tree pattern matching is used to determine which rules apply in any given case.

Typol [19, 34], the semantic description language used in Centaur [10], is based on natural semantics. A Typol description is a collection of axioms and inference rules. The general evaluation strategy is to compile the rules of the description into PROLOG rules and the resulting single equation to be proved into a PROLOG goal.

Natural semantics and Typol alone provide an elegant descriptive mechanism for many kinds of contextual constraints. The original PROLOG-based implementation, however, was inefficient since it was not incremental. Attali [3] addresses the problem of incremental evaluation in Typol by transforming a Typol definition into an attribute grammar, thereby providing access to methods for incremental attribute evaluation. Attali also shows how to implement partial evaluation of Typol programs and how to extend the approach to dynamic semantics. While the transformation from Typol to an attribute grammar achieves incremental evaluation, the attribute values manipulated by the attribute grammar are not necessarily modifiable incrementally. This means that, without care, the entire symbol table for a program will be treated as a unit unless techniques like Hoover and Teitelbaum's [30] are adopted. Moreover, by adopting attribute grammars, one adopts not only their strengths but also their weaknesses.

Like attribute grammars, LCGs are primarily operational descriptions of contextual constraints and could be used as implementation vehicles for higher-level description techniques. Like context relations, LCGs allow one to focus on the creation and use of derived information in a database setting, rather than on the flow and interaction of information within the evaluator. Nonlocal interactions among structures in a document are automatically handled via database interactions. The author of an LCG description is allowed to specify both the structure and the content of the logic database. The information in the database can be made available to users via the support facilities discussed in Section 6.

5.1 Incremental Evaluation

For a given language description, satisfying some goals requires satisfying other goals. Many interactions between goals act through the logic database. In other cases, the flow of contexts or other data from one subtree to another can be locally determined. An LCG evaluator may use, but does not require, knowledge of dependencies between goals. Naturally, if the user of an LCG system takes advantage of the evaluation strategies employed by the system, the performance of the evaluator can be enhanced.

Inconsistencies between documents and their derived information arising from incremental changes are detected by a consistency manager, a simple reason maintenance system. When an inconsistency is detected, the consistency manager determines which derived data must be removed and which goals have to be reattempted. Incremental evaluation continues until consistency is restored or until a circularity is suspected.

Careful selection of goals to be retried after database modifications is crucial to efficient incremental evaluation. Removal of data from the database, the simpler of the two cases, is handled using dependency-directed backtracking [58]. The evaluator records which data are used to satisfy each goal. When a datum is removed, the consistency manager retries all those goals whose satisfaction depended on it.

Two different ways to handle additions to the database were developed: holes and shadowing rules. *Holes* provide a means for representing data whose absence from the database was used in satisfying a goal. When a hole is filled, the consistency manager reattempts all of the goals that depended on the absence of that datum. Holes are computationally efficient, but memory intensive. The strategy of using holes does not scale well when many sparse collections are searched for a datum. Thus, holes are best suited for situations in which the data whose absence they represent would normally be present.

Shadowing rules are inference rules, computed by static analysis of an LCG description, that help to determine which goals must be attempted again when a datum is added to the database. Shadowing rules require less storage but more computation than holes, making them better suited for situations in which data sparsely populate the database. The presence of a static analyzer simplifies descriptions and relieves the authors of language descriptions from specifying many details.

5.2 LCGs and Logic Programming

Making LCGs practical required several modifications to the basic PROLOG model of logic programming [59]:

Partitioned database. The logic database is explicitly structured into *collections* of *data tuples*. Collections and data tuples are first-class objects: They can be created and destroyed dynamically. Collections are created and data tuples are added to collections as a side effect of satisfying a goal. Data tuples can refer to collections but cannot contain unbound logical variables.

Partitioning the logic database into collections improves the performance of an incremental evaluator while allowing the author of a language description to express directly the partitionings often found in languages. For example, collections can be used to represent scopes in a programming language.

Distinction between code and data. Terms that can appear in the database must be distinct from terms that can appear as the heads of procedure clauses. This limitation assures that an entire collection of goals and procedures can be statically analyzed.[9]

Only ground terms may be added or removed from the database. This assures that all changes to the logic database will be explicit.

Ownership. Every collection and data tuple in the database are owned by one or more subtrees in the document being edited. Collections are permanently associated with their owning subtree. The tuples in a collection may change repeatedly, but the collection itself retains its existence and identity until its owning subtree is destroyed. Ownership is used by the consistency manager to relate changes in the underlying document to changes in the information being maintained.

Contexts. Each subtree in a document has zero or more associated collections of tuples, called its *contexts*. All goals associated with that subtree are evaluated relative to the subtree's contexts. The contexts of a subtree are determined dynamically.

The primary use for a context is to provide access from the abstract syntax tree to other collections. A secondary use is to propagate information locally among subtrees of the abstract syntax tree, similar to the methods developed for the Ergo system [47].

Separating the context from the goals in an LCG helps the author of a language description to focus on the essentials of each. Goals are defined relative to contexts; contexts must contain the information necessary to satisfy or disprove their goals.

Ordering among goals. The ordering among goals is not formally specified, so standard PROLOG programming techniques that rely on known orderings among data tuples in the database may not apply. In particular, the use of **assert** and **retract** in an LCG differs from their use in PROLOG.

5.3 Example

Figure 5 shows a very simple LCG for a language that requires that each name be defined before it is used. Names are defined by either declarations or importation; names are used in generic "uses" and in procedure calls. The notation resembles the *Colander* language. An identifier prefixed by "$" denotes a node in the abstract syntax tree; "$$" denotes the node associated with the goal currently being satisfied. Identifiers prefixed by "*?*" denote logical variables; the anonymous logical variable is denoted by *??*. The notation ⟨*?Form, ?Collection*⟩ indicates that *?Form* is to be evaluated with respect to the given collection. An "entity" is a special kind of collection that can be used to represent linguistic objects such as variables. The first three sections of the LCG define the facts and procedures used in the specification.

[9] Extending the analysis to handle the dynamic addition of new goals or clauses to the logic program is straightforward in principle but costly in practice. It requires a new level of dependency-directed backtracking relating the inputs and outputs of the static analysis.

```
Definition-Specific Data Tuples:
    declared (?Name, ?Entity)              /* Fact representing binding of ?Name to ?Entity */
    imported (?Name, ?Entity)              /* Fact representing importation of ?Name as ?Entity */
    enclosing-scope (?Scope)               /* Fact representing the next outer scope */
    type-of (?Type-mark)                   /* Entity property representing the type
                                                    of an entity—either "id" or "proc" */
Definition-Specific Procedures:
    < visible (?Name, ?Entity), ?Scope > - < declared (?Name, ?Entity), ?Scope >
    < visible (?Name, ?Entity), ?Scope > - < imported (?Name, ?Entity), ?Scope >

    < lookup (?Form), ?Scope > - < ?Form, ?Scope >, '
    < lookup (?Form), ?Scope > - < enclosing-scope (?Scope_1), ?Scope >,
                                 < lookup (?Form), ?Scope_1 >
Builtin Procedures:
    assert (< ?Tuple, ?Collection >)       /* Adds ?Tuple to ?Collection */
    context (?Loc, ?Scope)                 /* Binds ?Scope to single context of ?Loc */
    new-entity (?Entity)                   /* Binds ?Entity to unique marker */
    new-context* (?Context)                /* Binds ?Context to a new context and
                                                    propagates ?Context to all subtrees */
    not (?Form)                            /* Suceeds if and only if ?Form fails */
    string-name (?Loc, ?Name)              /* Binds ?Name to string name of ?Loc */
Grammar:
⟨document⟩ → ⟨def⟩* ⟨use⟩*
⟨def⟩        → "DEF" id
             - context ($$, ?Scope),
               string-name ($id, ?Name),
               not (< visible (?Name, ??), ?Scope >)
                   "Invalid redeclaration of ?Name in this scope",
               new-entity (?Entity),
               assert (< type-of ("id"), ?Entity >),
               assert (< declared (?Name, ?Entity), ?Scope >).
⟨def⟩        → "IMPORT" id                       /* Similar to goals for "DEF" above */
⟨def⟩        → "PROC" id ⟨def⟩* ⟨use⟩*
             - context ($$, ?Context),
               new-context* (?New-context),   /* Create new context and propagate to children */
               assert (< enclosing-scope (?Context), ?New-context >)
             -                                          /* Declare id as a procedure */
⟨use⟩        → "USE" id
             .- context ($$, ?Scope),
               string-name ($id, ?Name),
               < lookup (visible (?Name, ?Entity)), ?Scope >
                   "No identifier named ?Name could be located",
               < type-of ("id"), ?Entity >   "?Name not declared as an identifier"
⟨use⟩        → "CALL" id                   /* Check that id is declared as a procedure */
```

Fig. 5. Fragment of a simple logical constraint grammar.

The goal associated with the production [[⟨*def*⟩ →"DEF" **id**]] first accesses the current context and binds the external (string) name of the identifier using the procedure **string-name**. The goal then checks that no other binding to that name occurs in the current scope. If another binding is present, the **not** term fails, and an error message will be created. Otherwise, the goal creates a new entity that will represent the variable being declared, sets the type of the entity to **'id'**, and adds a fact that will represent the new binding to the current scope.

The subgoal ⟨**lookup(visible(***?Name, ?Entity***))**, *?Scope*⟩, appearing in the goal associated with ⟦⟨*use*⟩ →"USE" **id**⟧, searches for a binding involving the name of the identifier appearing in the production. Once again, the primitive **string-name** is used to get the actual external name of the identifier. The term **lookup** implements nested block structure by recursively traversing outward through nested scopes, evaluating the procedure **visible** within each scope. The use of the cut operator ("!") in **lookup** ensures that only the first solution to *?Form* will be investigated. If the subgoal involving **lookup** fails, an error message is generated. If it succeeds, then *?Entity* will be bound to the entity located in the search; the **type-of** property can then be validated.

In the goal associated with ⟦⟨*def*⟩ →"PROC" **id** ⟨*def*⟩*⟨*use*⟩*⟧, the built-in function **new-context*** is used to create and propagate a new collection to all of the direct descendants in the internal tree. *Colander* also provides a primitive that allows greater control over which subtrees inherit the newly created collection.

One simple extension to this example is to maintain a call-graph of the program by adding a goal associated with the production ⟦⟨*use*⟩→"CALL" **id**⟧ that adds facts of the form "**calls(***?Proc1, ?Proc2***)**" into some collection. Database triggers could then be used to update a graphical view of the call graph incrementally.

5.4 An Implementation: *Colander*

The *Colander* language, one of *Pan*'s formalisms for language description, embodies the LCG approach. The language and its run-time support extend the basic approach in ways that improve either the efficiency of the description, the usability of the description language, or both.

Pass-structured evaluation. *Colander* partitions the goals associated with a syntactic structure into two classes: those goals whose primary use is to establish the context used by that structure or by its substructures, and those goals whose primary use is to express a contextual constraint. The classification of goals is indicated explicitly in a *Colander* description. Goals in the first class are called *first-pass* goals. They are evaluated during a top-down preorder walk of the internal tree. Goals in the second class are called *second-pass* goals. They are evaluated after the first-pass goals and, in the current implementation, are evaluated after the first-pass goals of the subtree's children have been evaluated. For example, the first goal associated with the production ⟦⟨*def*⟩ →"PROC" **id** ⟨*def*⟩*⟨*use*⟩*⟧ should be a first-pass goal, while the second goal on that production could be a second pass goal. In general, goals that establish contexts should be first-pass goals; all others will be second-pass goals.

Multiple kinds of collections and data tuples. *Colander* distinguishes three kinds of collections, each holding its own kinds of data. *Datapools* are collections of *facts*. Datapools are used to aggregate facts that can or should be treated as a single unit. There can be multiple instances of the same fact. For example, each scope in a program might be represented using a separate datapool containing facts about the declarations appearing in that scope, together with data relating that scope to the other scopes in the program.

An *entity* is a collection that can be used to represent objects of the described language. Entities can be used to hold the attributes of a particular object such as a variable, a procedure, or a paragraph. Information about entities is represented using *entity properties*. A *property* is a named value associated with a collection. Properties are single-valued; unlike facts, it is not possible for a collection to contain more than one property value with a given property name.

Subtrees can also be considered as collections holding *subtree properties*. Subtrees are created and destroyed by *Ladle* during syntactic analysis. Structural information about the internal tree is represented using subtree properties.

Maintained subtree properties. *Maintained subtree properties* are like attributes in an attribute grammar. Their values are defined by local procedures associated with the grammar production that defines the subtree. The value of a maintained property is computed relative to the subtree for which the property is being defined and can depend on any other values available, including the properties of the parent, child, or immediate sibling subtrees.

Maintained properties are calculated on demand. Once the value of a maintained property is computed, it is stored in the logic database. The consistency manager is then responsible for recomputing the value of a maintained property as necessary.

A typical use for maintained subtree properties is in determining the type of an expression. The value of the type would be stored in the maintained property, and the subtree-specific procedure would define how to compute the type as a function of the subtree's subexpressions.

Although an attribute grammar can be emulated directly by an LCG using only maintained subtree properties, it is far more efficient to use the database and the context for moving values through the tree. In most attribute grammar descriptions, inherited attributes either summarize relatively local structural information about the internal tree or else consist of a "symbol table" containing nonlocal information. *Colander* subsumes the "symbol table" into the database along with other information about the tree. Local structural information can be passed like inherited attributes by creating and propagating new context datapools containing that information. Synthesized values propagate from leaves toward the root, as in attribute grammars; these appear frequently in *Colander* descriptions as maintained properties.

Client properties. A *client property* is a subtree property that is neither a structural property nor a maintained property. Client properties are usually manipulated by programs or components via a client interface, although they can appear in a *Colander* description. Client properties are not under consistency maintenance unless they are declared and used within the language description. One way in which we have used client properties is in the prototype of a context-based prettyprinter. The prettyprinter attaches its own information to subtree nodes using client properties.

Database triggers. *Colander* provides triggers that are activated when data are added or removed from the database. Triggers provide a uniform mechanism for implementing notifier functions used by clients. They are used internally as well, for example, to implement shadowing rules.

Messages to the user. Any term appearing in a goal or a procedure body can be suffixed with a message. When a goal fails during evaluation, the message associated with the most recent term to fail is captured and used as described in Section 6.2.

Special primitives. The internal tree used in *Pan* allows subtrees with an arbitrary number of children called *sequence nodes*. *Colander* provides several functions for mapping goals over the children of a sequence subtree. *Colander* also provides two special functions that interact with the consistency manager. The function **all-solutions** is like Prolog's **bagof** operator. It can be used to calculate all solutions to a goal, assuming that the goal terminates. It is reevaluated whenever the set of solutions might have changed. A typical use for **all-solutions** is to gather up all of the data tuples matching a particular query. For example, **all-solutions** might be used to obtain all of the bindings to a particular name in order to discriminate among the possible definitions of an overloaded operator. The function **notever** is a form of negation that is monitored by the consistency manager. If a new solution arises that might cause the **not** to fail, then the goal containing the **notever** will be retried.

6. USER SERVICES

Pan's ultimate purpose is to assist its intended users. This section discusses some of the technical problems associated with providing coherent user services. A more thorough treatment of *Pan*'s approach to delivering language-based technology appears elsewhere [66].

6.1 Document Models

User and system must communicate about what is being edited. Language-based editors often present a document model based implicitly on internal representations, and the abstract syntax tree is sometimes proposed as a "natural" model for user interaction. In practice, however, the design of a tree representation (the abstract grammar in *Pan*) is strongly influenced by internal clients of the data (e.g., a *Colander* specification). These influences are unrelated to the way users understand document structure.

Pan decouples internal representation from user interaction. The language description mechanism provides a loose framework in which the author of each description is expected to design a model of document structure that is appropriate for the language, its intended users, and their tasks. Alternate descriptions for a single underlying language can provide different services based on different models, suitable for different users and tasks. For example, one might provide different operand classes and more elaborate error handling for novices than for experienced users, or different query services for reengineering than for authoring.

This framework is based on two assumptions about how people understand document structure: they think in terms of structural components instead of trees, and they think of those components in the specific terminology of particular languages. To most users, a "statement" is just a "statement"; it is neither an "operator" nor a "subtree."

Operand classes. *Pan*'s primary mechanism for hiding internal document representation is the *operand class*. Operand classes are arbitrary, possibly overlapping, named collections of document components. They form the basis for structure-oriented selection, navigation, highlighting, and editing. Class membership is dynamic, based on class definition and the current database of derived information.

The relationship between tree operators and operand classes is user-oriented and many-to-many, in contrast to the use of operators and phyla [35] for tree specification. Operators not in any operand class are hidden from users and are not part of the structural model. In the Synthesizer Generator, a similar effect is achieved by the "resting place" mechanism [52], but there is only one (anonymous) class of resting places and its specification is embedded in the unparsing scheme rather than being dynamically determined. Operand classes define "views" of documents by controlling access to underlying structure, an approach that is independent of and complementary to document display as exemplified by Garlan's unparsing [24] and Reiss's views [51].

Class definition. With a few built-in exceptions,[10] operand classes are defined by predicates on nodes of the internal tree representation. A new class may be created at any time using a declarative syntax that specifies the class name, the defining predicate, and options. One particularly useful option specifies daemons to be invoked before or after any structural selection based on the class.

Several language-independent classes are predefined. For example, '**Syntactic Error**'[11] and '**Unsatisfied Constraint**' denote the sets of syntactic and contextual-constraint variances, respectively. '**Language Error**', defined to be the union of '**Syntactic Error**' and '**Unsatisfied Constraint**', is more appropriate when the distinction would be irrelevant or confusing to the user. The class '**Query Result**' permits perusal of a set of nodes returned by the most recent database query.

Each language description adds language-specific classes, starting with structural ones such as '**Expression**', '**Statement**', '**Declaration**', and

[10] The classes '**Character**', '**Word**', and '**Line**' are defined in terms of the text stream, and '**Lexeme**' in terms of the lexical stream; all offer services similar to the structural classes.
[11] Although 'syntax errors' are just a kind of variance in the *Pan* system, to users they are designated by their traditional name.

‘**Procedure**’. Since classes may overlap, a node representing a malformed statement might be in both the ‘**Statement**’ and ‘**Syntax Error**’ classes.

Finer distinctions are possible. In one description, ‘**Language Error**’ contains all syntactic variances plus those unsatisfied constraints concerning the language definition; other unsatisfied constraints are in the class ‘**Stylistic Violation**’.

Diagnostics. Each language description associates diagnostic messages with potential variances. The presence of variances in a document precipitates no special action other than the appearance of designated panel flags. The user requests more information by invoking appropriate services.

Pan’s other services take no particular notice of variances. Malformed statements, statements with unsatisfied constraints, and sometimes even statements within malformed blocks can still be treated as statements. Ill-formed and well-formed documents need not appear much different.

6.2 Using Derived Information

Text editing is so fundamental in *Pan* that it might not be apparent at all when language-based information has been derived. *Pan*’s default configuration specifies panel flags that appear when this is the case, as shown in Figure 6. Derived information is exploited only through specific services, all optional and under user control. This section describes a few such services, deferring discussion of language-based editing to Section 6.4.

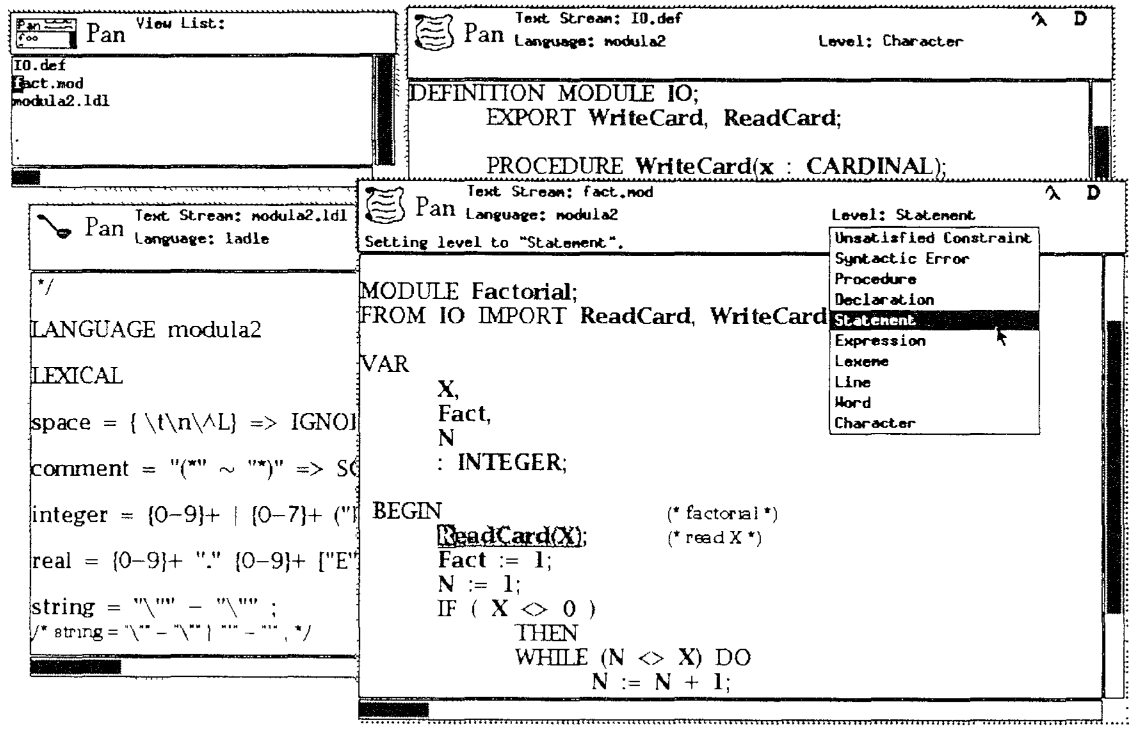
Presentation enhancements. Visual text attributes draw attention to particular document components. For example, *font shifts* reveal lexical categories: keywords, identifiers, comments, and unanalyzed text. This use of typography contributes significantly to program readability in our experience, but only when the font selection is tuned for each language.

Prettyprinting reindents documents to reveal syntactic structure. More advanced forms of prettyprinting for program documents, including semantically driven elision, are under development.

Structural highlighting may be designated for text in particular categories, varying the color of both text and background. Useful examples include the results of the most recent database query and different categories of variances. Although highlighting carries less information than diagnostic messages, experienced programmers often diagnose simple variances at a glance, once attention is drawn to them.

The operand level. Each *Pan* window has a current *operand level*, chosen by the user from a menu of classes specified by the language description (see Figure 6).[12] The operand level is a weak input mode that modulates the operation of generic commands like **Cursor-Forward**, **Cursor-Backward**, **Cursor-To-First**, **Cursor-To-Last**, **Cursor-In**, **Cursor-Out**, **Cursor-Search-Forward**, **Select**, and **Delete**. The operand level affects no other commands and never affects text-oriented editing.

[12] As with most menu-based services, equivalent keyboard commands are available.



```
IO.def
fact.mod
modula2.ldl
```

```
DEFINITION MODULE IO;
        EXPORT WriteCard, ReadCard;

        PROCEDURE WriteCard(x : CARDINAL);
```

```
*/
LANGUAGE modula2

LEXICAL

space = { \t\n\^L} => IGNO

comment = "(*" ~ "*)" => S

integer = {0-9}+ | {0-7}+ ("

real = {0-9}+ "." {0-9}+ ["E"

string = "\"" - "\"" ;
/* string = "\" - "\"" | "" - "" , */
```

```
MODULE Factorial;
FROM IO IMPORT ReadCard, WriteCard

VAR
        X,
        Fact,
        N
        : INTEGER;

BEGIN                               (* factorial *)
        ReadCard(X);                (* read X *)
        Fact := 1;
        N := 1;
        IF ( X <> 0 )
                THEN
                WHILE (N <> X) DO
                        N := N + 1;
```

Fig. 6. Editing a program using *Pan*.

Structural navigation. The operand level enables language-specific navigation. For example, when the current level is '**Statement**', a press of the left mouse button selects the "nearest" (based on a heuristic) component defined to be in that class. Generic navigation commands perform various tree walks, selecting only components in the class.

Operand classes associated with variances are usually configured (by specification of an appropriate after-daemon) to announce the diagnostic associated with each member as it is selected. Structural navigation then supports the location and diagnostic of each variance in turn.

6.3 Inconsistency and Reanalysis

Any situation where one kind of information is derived from another invites inconsistency between the two. The syntax-recognizing approach, where information is derived from text, is no exception. One aspect of the problem arises during text-oriented editing, when derived information may disagree with what the user sees. For example, font shifts would be incorrect immediately after the textual transformation of a statement into a comment. In most cases, however, presentation enhancements remain *almost* correct in ways easily understood by experienced users. Designated panel flags (in Figure 6, a treelike glyph for syntax and '**D**' for the Colander database) appear pale during inconsistency.

To avoid more serious confusion, *Pan*'s *automatic reanalysis* policy ensures that language-based interaction takes place *only* when text and derived information are consistent. Should the user invoke such a command during inconsistency, *Pan* triggers incremental analysis before attempting it. Since analysis (almost) always succeeds, this policy does not restrict the user, although it may cause delay. Possible relaxations of this policy are being explored, where more language-based services might usefully work on an *almost correct* basis using inconsistent information.

Pan's lazy reanalysis policy presumes that the user understands the general state of the document and can judge trade-offs. Incremental analysis takes place only when requested, either *implicitly* by invocation of an operation that triggers automatic reanalysis or *explicitly* by invocation of **Analyze-Changes**. Nothing prevents a *Pan* user from typing an entire document without analysis. *Pan* is designed to encourage frequent analysis by making it cost-effective to the user.

6.4 Mixed-Mode Editing

Pan's approach to language-based editing is to broaden options, not to narrow them. The user should be able to edit textually at any time and at any place in the document presentation; it should be equally possible to edit in terms of derived information at any time and at any place.

A dual aspect cursor. *Pan* commands, text- or structure-oriented, may be invoked without prerequisite. Two mechanisms make this work. The first, automatic reanalysis, ensures that derived information is consistent with the text for any operations that require it. The second is *Pan*'s dual aspect edit cursor.

Pan's edit cursor always has a textual location, displayed as an inverted box. It may also have a location corresponding to some structural component, as it does in Figure 6 where the cursor is at a '**Statement**' (revealed by colored background shading). Setting the structural cursor also selects the component textually and positions the text cursor at its beginning.

Any editing operation that requires a cursor location uses the appropriate aspect: text or structure. If the cursor has no structural aspect, then one is inferred from the text cursor's location by the same mechanism used when the user selects a structural component by pointing with the mouse. This design resolves the "point versus extended cursor" problem [62] by providing both behaviors simultaneously.

Simple editing. No user commands in the prototype implementation modify internal document structure directly. **Cut**, when invoked with a structural selection (as in Figure 6), achieves the same effect by moving the associated text out of the stream and into the clipboard. The internal representation of the deleted component persists until the next reanalysis, but it is invisible because automatic reanalysis will remove it before any commands can use it. **Paste** inserts text from the clipboard. If the context is appropriate, subsequent incremental analysis derives equivalent structural information quickly.

When a structurally inspired **Cut** and **Paste** sequence violates the underlying language definition, the operations succeed anyway. Problems are diagnosed by precisely the same mechanisms that handle other variances, and the user is free to continue. By discarding derived information, this approach increases analysis time slightly. On the other hand, it guarantees the well formedness of the internal representation, since the language definition is already built into *Pan*'s parser.

Complex mechanisms for direct structural modification can confuse users; editing operations may fail for the kinds of reasons *Pan* hides. For example, it might seem reasonable to copy the formal parameter list of a procedure definition and paste it into a procedure call. Although the two lists of identifiers could be textually identically and conceptually related from a user's perspective, implementation concerns may dictate incompatible internal representations. General solutions to this problem involve guessing the user's intent, an approach avoided in *Pan*.

Another cost of *Pan*'s textual approach is the loss of any nonderivable annotations on document components during structural **Cut** and **Paste**. We believe this can be repaired by the following (as yet unimplemented) expedient: cut both text and structure, paste text, reanalyze, and copy nonderivable data only when the new structure is isomorphic to the old.

Other language-based operations. The ultimate advantage of language-oriented editing lies in an open-ended collection of services that draw upon a rich repository of information to assist users in commonly performed tasks. Some services locate and diagnose variances, as described above. This section describes other simple examples, emphasizing first that services are implemented by combining generic and language-specific components, and second that the user need not be aware of complex internal representations that make them work.

One of the few forms of query supported by ordinary text editors is textual search. Searching in *Pan* can draw upon *any* derived information. For example, one command locates the declaration and all uses of a programming language variable, which the user identifies by pointing. The results of this and other language-based queries are made available by an interface similar to the one used for variances. Text associated with components of the current query is highlighted, and the user may set the operand level to "**Query Result**" to navigate through the components. The query itself is defined as part of the language description and supports other services, for example, a command that moves the cursor to the declaration of a specified variable. The latter command is an example of navigation through hypertext-like links, defined by the underlying language and recorded in the database during analysis. Like all powerful text editors, *Pan* supports textual replacement based on regular expression matching. But users sometimes want replacement to depend on language structure rather than on textual structure, even when the two are similar. For example, whole-word replacement (where replacing substrings of longer words is not desired) in natural language documents is difficult to specify using patterns. One variant of *Pan*'s replacement command matches patterns only against words (lexemes), as defined by the particular language. Another variant renames variable instances in programs, drawing upon information in *Pan*'s database to avoid renaming enclosed variable definitions that have the same lexical name but that are logically different variables.

7. RETROSPECTIVE

Pan has limitations with respect to our long-range vision: current description techniques are aimed at a particular class of formal languages; the implementation supports only one language per document; a single analysis may span multiple documents, but only within one language; the system provides only part of the desired flexibility in generating visual presentations. Support for novice programmers and learning environments, support for program execution, graphical display and editing, and the use of a persistent software development database are not supported in the current system. The *Ensemble* project, which will create a successor to *Pan*, is addressing some of these issues.

Ongoing research projects are using the leverage gained from *Pan*. Projects near completion include the development of advanced user-interaction techniques, including the use of *Colander* to specify and control user-centered program viewing [65], the development of new language descriptions, and investigations into ways to strengthen *Pan*'s language description techniques [23].

With the exception of a simple noneditable tree display, the presentations in *Pan* are all textual and are all closely coupled to the concrete syntax descriptions of the documents. The *Ensemble* project is generalizing *Pan*'s approach in three ways:

(1) much richer mappings among document structure, presentations, and specification of appearance, building heavily on the experience gained from the VORTEX document system [13, 14];
(2) the extension of editing and viewing to a wide range of media—text, graphics, sound, and video; and
(3) integrated support for compound documents.

The notation of logical constraint grammars, being based on clausal logic, has proved to be quite effective for expressing queries against the database. Complete descriptions of languages, however, rapidly become verbose. One approach to remedying this situation is to use the LCG mechanism as an implementation vehicle for higher-level semantic descriptions, such as those based on Natural Semantics [34].

ACKNOWLEDGMENTS

Pan is the work of many individuals besides the authors. Christina Black, Jacob Butcher, Bruce Forstall, Mark Hastings, and Darrin Lane have all made substantial contributions. The encouragement, suggestions, and support of Bill Scherlis have played a major role as well. We are grateful to Yuval Peduel and the reviewers for helpful comments on earlier drafts of this paper. Finally, we wish to acknowledge the community of "editor people" with whom we have interacted over the past several years. Many of their best ideas appear somewhere within *Pan*.

REFERENCES

1. ACM. Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulatıon. *SIGPLAN Not.* (ACM) *16*, 6 (June 1981).
2 AMBRAS, J., AND O'DAY, V. Microscope: A knowledge-based programming environment *IEEE Softw. 5*, 3 (May 1988), 50-58.
3 ATTALI, I. Compiling TYPOL with attribute grammars In *Programmıng Languages Implementation and Logic Programmıng*, P. Deransart, B. Lorho, and J. Maluszński, Eds. Lecture Notes in Computer Science, vol. 348. Springer-Verlag, New York, 1988, pp. 252-272.
4. BAECKER, R. M , AND MARCUS, A. *Human Factors and Typography for More Readable Programs*. ACM Press, New York, 1990
5. BAHLKE, R., AND SNELTING, G. The PSG system: From formal language definitions to interactive programmıng environments *ACM Trans. Program Lang. Syst 8*, 4 (Oct 1986), 547-576.
6. BALLANCE, R. A. Syntactic and semantic checking in language-based editing systems. Ph D. dıssertation, Computer Science Division—EECS, Univ of Calıfornia, Berkeley, Dec. 1989. (Available as Tech. Rep. UCB/CSD 89/548.)
7 BALLANCE, R. A., AND GRAHAM, S L. Incremental consistency maintenance for interactive applicatıons. In *Proceedıngs of the 8th Internatıonal Conference on Logıc Programming*, K Furukawa. Ed. MIT Press, Cambridge, Mass , 1991, pp 895-909.
8. BALLANCE, R A., BUTCHER, J., AND GRAHAM, S. L. Grammatical abstraction and incremental syntax analysıs in a language-based editor In Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementatıon. SIGPLAN *Not.* (ACM) *23*, 7 (July 1988), 185-198.

9. BALLANCE, R. A., VAN DE VANTER, M. L., AND GRAHAM, S. L. The architecture of Pan I. Tech. Rep. UCB/CSD 88/409, Computer Science Division—EECS, Univ. of California, Berkely, Mar. 1988.
10. BORRAS, P., CLÉMENT, D., DESPEYROUX, T., INCERPI, J., KAHN, G., LANG, B., AND PASCUAL, V CENTAUR: The system. In [27], pp. 14-24.
11. BUDINSKY, F. J., HOLT, R. C., AND ZAKY, S. G. SRE—A syntax-recognizing editor. *Softw. Pract. Exper. 15*, 5 (May 1985), 489-497.
12. BUTCHER, J. Ladle. Master's thesis, Computer Science Division—EECS, Univ. of California, Berkeley, Nov. 1989. (Available as Tech. Rep. UCB/CSD 89/519.)
13. CHEN, P., AND HARRISON, M. A. Multiple representation document development. *Computer 21*, 1 (Jan. 1988), 15-31.
14. CHEN, P., COKER, J., HARRISON, M. A., MCCARRELL, J., AND PROCTER, S. The VORTEX document preparation environment. In *Proceedings of the 2nd European Conference on TEX for Scientific Documentation*, J. Desarménien, Ed., Lecture Notes in Computer Science, vol. 236. Springer-Verlag, New York, 1986, pp. 45-54.
15. COHEN, J. Constraint logic programming languages. *Commun. ACM 33*, 7 (July 1990), 52-68.
16. CONRADI, R., DIDRIKSEN, T. M., AND WANVIK, D., EDS. *Advanced Programming Environments*. Lecture Notes in Computer Science, vol. 244. Springer-Verlag. New York, 1986.
17. CORBETT, R. P. Static semantics and compiler error recovery. Ph.D. dissertation, Computer Science Division—EECS, Univ. of California, Berkeley, June 1985. (Available as Tech. Rep. UCB/CSD 85/251.)
18. DERANSART, P., JOURDAN, M., AND LORHO, B. *Attribute Grammars: Definitions, Systems, and Bibliography*. Lecture Notes in Computer Science, vol. 323. Springer-Verlag, New York, 1988.
19. DESPEYROUX, T. Executable specification of static semantics. In *Semantics of Data Types*, G. Kahn, D. B. MacQueen, and G. D. Plotkin, Eds. Lecture Notes in Computer Science, vol. 173. Springer-Verlag, New York, 1984, pp. 215-233.
20. DONZEAU-GOUGE, V., HUET, G., KAHN, G., AND LANG, B. Programming environments based on structured editors: The MENTOR experience. In *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall, Eds. McGraw-Hill, New York, 1984, pp. 128-140.
21. DOWNS, L. M., AND VAN DE VANTER, M. L. Pan I version 4.0: An introduction for users. Tech. Rep. UCB/CSD 91/659, Computer Science Division—EECS, Univ. of California, Berkeley, Nov. 1991.
22. DOYLE, J. A truth maintenance system. In *Readings in Artificial Intelligence*, B. L. Webber and N. J. Nilsson, Eds. Tioga, Palo Alto, Calif., 1981, pp. 496-516.
23. FORSTALL, B. T. Experience with language description mechanisms in Pan. Master's thesis, Computer Science Division—EECS, Univ. of California, Berkeley, Nov. 1991.
24. GARLAN, D. Flexible unparsing in a structure editing environment. Tech. Rep. CMU-CS-85-129, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., Apr. 1985.
25. GOLDBERG, A. Programmer as reader. *IEEE Softw. 4*, 5 (Sept. 1987), 62-70.
26. HABERMANN, A. N., AND NOTKIN, D. Gandalf: Software development environments. *IEEE Trans. Softw. Eng. SE-12*, 12 (Dec. 1986), 1117-1127.
27. HENDERSON, P., ED. *ACM SIGSOFT 88: 3rd Symposium on Software Development Environments* (Boston, Nov. 28-30, 1988). ACM, New York, 1988.
28. HILFINGER, P. N., AND COLELLA, P. Fidil: A language for scientific programming. In *Symbolic Computation: Applications to Scientific Computing*, R. Grossman, Ed. SIAM, Philadelphia, Pa., 1989, pp. 97-138.
29. HOLT, R. W., BOEHM-DAVIS, D. A., AND SCHULTZ, A. C. Mental representations of programs for student and professional programmers. In *Empirical Studies of Programmers: Second Workshop*, G. M. Olson, S. Sheppard, and E. Soloway, Eds. Ablex, Norwood, N.J., 1987, p. 33.
30. HOOVER, R., AND TEITELBAUM, T. Efficient incremental evaluation of aggregate values in attribute grammars. In Proceedings of the SIGPLAN 86 Symposium on Compiler Construction. *SIGPLAN Not.* (ACM) *21*, 7 (July 1986), 39-50.

31. Horton, M. R. Design of a multi-language editor with static error detection capabilities. Ph.D dissertation, Computer Science Division—EECS, Univ. of California, Berkeley, 1981.
32. Horwitz, S., and Teitelbaum, T Generating editing environments based on relations and attributes. *ACM Trans. Program. Lang. Syst. 8*, 4 (Oct. 1986), 577-608.
33. Jalili, F., and Gallier, J H. Building friendly parsers. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages* (Albuquerque, N.M , Jan. 25-27). ACM, New York, 1982, pp. 196-206.
34. Kahn, G. Natural semantics. Tech. Rep. 601, INRIA, Feb. 1987.
35. Kahn, G., Lang, B., Mélèse, B., and Morcos, E Metal: A formalism to specify formalisms *Sci. Comput. Program. 3*, 2 (Aug 1983), 151-188.
36. Kaiser, G. E. Semantics for structure editing environments. Ph.D. dissertation, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., May 1985.
37. Kirslis, P. A C. The SAGA editor: A language-oriented editor based on an incremental LR(1) parser. Ph.D. dissertation, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Dec. 1985.
38. Knuth, D. E. Literate programming *Computer J. 27*, 2 (May 1984), 97-111.
39. Lampson, B. W. *Bravo Users Manual*. Palo Alto, 1978.
40. Lang, B. On the usefulness of syntax directed editors. In [16], pp 47-51.
41. Letovsky, S. Cognitive processes in program comprehension In *Empirical Studies of Programmers*, E. Soloway and S. Iyengar, Eds. Ablex, Norwood, N J., 1986, pp. 58-79
42. Letovsky, S., and Soloway, E. Delocalized plans and program comprehension. *IEEE Softw. 3*, 3 (May 1986), 41-49
43. Lewis, C., and Norman, D. A. Designing for error. In *User Centered System Design: New Perspectives on Human-Computer Interaction*, D. A. Norman and S. W. Draper, Eds. Erlbaum, Hillsdale, N J., 1986, pp. 411-432.
44. Masinter, L. M. Global program analysis in an interactive environment. Tech Rep. SSL-80-1, Xerox Palo Alto Research Center, Palo Alto, Calif., 1980.
45 Medina-Mora, R., and Feiler, P. H. An incremental programming environment *IEEE Trans. Softw. Eng. SE-7*, 5 (Sept. 1981), 472-481.
46. Neal, L. R. Human factors in computing systems and graphical interfaces. In *CHI + GI 1987 Conference Proceedings* (Toronto, Apr. 5-9, 1987). ACM, New York, 1987, pp. 99-102.
47 Nord, R. L., and Pfenning, F. The Ergo attribute system. In [27], pp. 110-120
48. Oman, P., and Cook, C. R. Typographic style is more than cosmetic. *Commun. ACM 33*, 5 (May 1990), 506-520
49. Plotkin, G D. A structural approach to operational semantics Tech. Rep. DAIMI FN-19, Computer Science Dept., Aarhus Univ., Aarhus, Denmark, Sept. 1981.
50 Prawitz, D. *Natural Deduction: A Proof-Theoretic Study*. Almquist and Wiksell, Stockholm, 1965.
51. Reiss, S. P. Graphical program development with PECAN program development system. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pa., Apr. 23-25, 1984). ACM, New York, 1984, pp. 30-41.
52. Reps, T., and Teitelbaum, T. *The Synthesizer Generator Reference Manual, Second Edition*. Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., 1987.
53. Reps, T , Teitelbaum, T., and Demers, A Incremental context dependent analysis for language based editors *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 449-477.
54 Rich, C., and Waters, R. C. The programmers apprentice: A research overview *Computer 21*, 11 (Nov. 1988), 10-25.
55. Smith, B., and Kelleher, G., Eds. *Reason Maintenance Systems and Their Applications* Series in Artificial Intelligence. Ellis Norwood, Chichester, 1988.
56. Soloway, E., and Ehrlich, K. Empirical studies of programming knowledge. *IEEE Trans. Softw. Eng. SE-10*, 5 (Sept. 1984), 595-609.
57. Stallman, R. M EMACS: The extensible, customizable, self-documenting display editor. In [1], pp. 147-156
58. Steele, G. L., Jr., and Sussman, G. J. Constraints. AI Memo 502, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Mass., Nov. 1978.

59. Sterling, L., and Shapiro, E. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, Mass., 1986.
60. Strömfors, O. Editing large programs using a structure-oriented text editor. In [16], pp. 39-46.
61. Teitelbaum, T., and Reps, T. The Cornell program synthesizer: A syntax-directed programming environment. *Commun. ACM 24*, 9 (Sept. 1981), 563-573.
62. Teitelbaum, T., Reps, T., and Horwitz, S. The why and wherefore of the Cornell program synthesizer. In [1], pp. 8-16.
63. Teitelman, W. A tour through Cedar. *IEEE Trans. Softw. Eng. SE-11*, 3 (Mar. 1985).
64. Van De Vanter, M. L. Error management and debugging in Pan I. Tech. Rep. UCB/CSD 89/554, Computer Science Division—EECS, Univ. of California, Berkeley, Dec. 1989.
65. Van De Vanter, M. L. User interface design for language-based editing systems. Ph.D. dissertation, Computer Science Division—EECS, Univ. of California, Berkeley. To be published.
66. Van De Vanter, M. L., Ballance, R. A., and Graham, S. L. Coherent user interfaces for language-based editing systems. *Int. J. Man-Mach. Stud.* (1992). To appear.
67. Waters, R. C. Program editors should not abandon text oriented commands. *SIGPLAN Not.* (ACM) *17*, 7 (July 1982), 39-46.
68. Winograd, T. Beyond programming languages. *Commun. ACM 22*, 7 (July 1979), 391-401.
69. Wood, S. R. Z—The 95% program editor. In [1], pp. 1-7.

Received January 1991; revised and accepted September 1991