The Pan Language-Based Editing System For Integrated Development Environments*

Robert A. Ballance† Susan L. Graham‡ Michael L. Van De Vanter‡

Abstract

Powerful editing systems for developing complex software documents are difficult to engineer. Besides requiring efficient incremental algorithms and complex data structures, such editors must integrate smoothly with the other tools in the environment, maintain a sharable database of information concerning the documents being edited, accommodate flexible editing styles, provide a consistent, coherent, and empowering user interface, and support individual variations and project-wide configurations. *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.

1 Introduction

The *Pan*[1] editing and browsing system originated from an investigation into ways to exploit language-based technology to benefit software developers. The *Pan* system rests on three fundamental assumptions.

1. Software is composed of textual documents having many structures that can be derived using language-based analysis. "Syntax" is just one such derived structure.
2. Developers are proficient with tools and languages; their ability to read and comprehend documents is the serious bottleneck.
3. The bridge between developers and their software will be intelligent editing interfaces.

Pan operates on *documents*—objects with both textual and structural aspects, such as formal designs and program components, as well as unstructured texts. To its users it appears as a fast, convenient, full-functioned text editor that happens to be extremely knowledgeable about the local working environment: the many languages in use, local conventions, and perhaps the user's own personal working habits. One might use *Pan* for editing text without ever giving a thought to its other capabilities. But at any time one might choose to broaden the dialogue with *Pan* to draw on many kinds of information (maintained by *Pan*) *about* the document. In particular, *Pan* can help reveal the multiplicity of structures, some user-defined, inherent in and among documents. *Pan* can be directed to use this information to guide editing actions, to configure and selectively highlight the textual display, to present answers to queries, and more.

The current implementation, *Pan I* [9], 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 I* is a multi-window, multiple-font, mouse-based editing system that is fully customizable and extensible in the spirit of Emacs [49].
- *Pan I* incrementally builds and maintains a collection of information about documents that can be shared with other tools.

*Research 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 No. 564516, by a gift from Apple Computer, Inc., and by the State of California MICRO Fellowship Program

†Department of Computer Science, University of New Mexico, Albuquerque, NM 87131. ballance@unmvax.cs.unm.edu

‡Computer Science Division (EECS), University of California, Berkeley, CA, 94720. graham@sequoia.berkeley.edu, mlvdv@sequoia.berkeley.edu

[1]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 tool box, the allusion to the lowly, ubiquitous kitchen utensil is apt. *Pan* is one of the PIPER projects at the University of California, Berkeley.

- *Pan I* users can freely mix text- and language-oriented manipulations in the same visual editing field; text editing is completely unrestricted.
- A single *Pan I* session may involve multiple languages.
- Adding new languages by writing language descriptions is just one of *Pan I*'s extension mechanisms.

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 [6, 13]. This correspondence permits a decorated abstract syntax tree, *Pan*'s primary structural representation, to be used directly for incremental LR parsing. A second technique, *logical constraint grammars*, adapts logic programming and consistency maintenance to the specification and enforcement of contextual constraints [5, 7]. Such constraints usually include the static semantic rules of a language, but can also include site-specific or project-specific restrictions such as naming conventions. Information gathered during constraint enforcement is retained in a logic database (available to other tools) and revised incrementally as documents change.

As part of our emphasis on *coherent user interfaces* [58], *Pan* language descriptions also include specifications that configure user interaction. For example, *operand classes* hide *Pan*'s internal tree representation behind a user-oriented conceptual model of document structure that can be tuned for each language. These specifications shield the user from *Pan*'s underlying technology, presenting instead a user interface that is designed to make the system's services as convenient and productive as possible.

Making *Pan*'s many mechanisms work together usefully demanded a number of system-wide design strategies. For example, it is axiomatic that documents being modified textually contain language errors more often than not. Every part of the system cooperates in support of *Pan*'s techniques for maximizing user service and minimizing information loss in the presence of errors.

This paper reviews the goals and early design decisions for *Pan* and surveys the implementation of the *Pan I* 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.

2 Design Rationale

The *Pan* project was motivated by a particular vision of the role to be played by language-based browsing and editing systems. This section describes that vision and shows how it defines the fundamental requirements for *Pan*'s design.

Throughout, the term *language-based* indicates that one or more of the facilities provided by the system makes 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. The term *editing interface* refers to the fact that those services are provided to the user through a generalization of the services of a traditional interactive editor.

2.1 The Working Environment

Pan is intended to support experienced professionals who manage large collections of interrelated documents. Many common assumptions about language-based editors, traditionally oriented toward novice authors, do not hold in this domain.

Understanding is the primary activity. Editors tuned for authoring fail to address today's problems. Software systems have become so large and complex that developers spend far more time trying to read, understand, modify, and adapt documents than they do creating them in the first place [23, 60]. A successful interactive development environment must support understanding by recognizing, exploiting, and making visible complex relationships within and among documents [56]. Related, but different support must be available for authoring and modifying documents.

There are many languages. Software developers use many formal languages: design languages, specification languages, structured-documentation languages, programming languages, and numerous small languages for scripts, schemas, and mail messages. Furthermore, programs often contain embedded "little languages" that impose their own conventions. For example, many subroutine libraries define mini-languages for long and complex argument sequences[2].

Languages and usage change. New languages arise, as do extensions and modifications to existing ones. Further, the way people use languages evolve, as community wide, personal, and project-specific conventions come and go. Finally, the services provided by

[2]Libraries for window systems often have these kinds of interfaces.

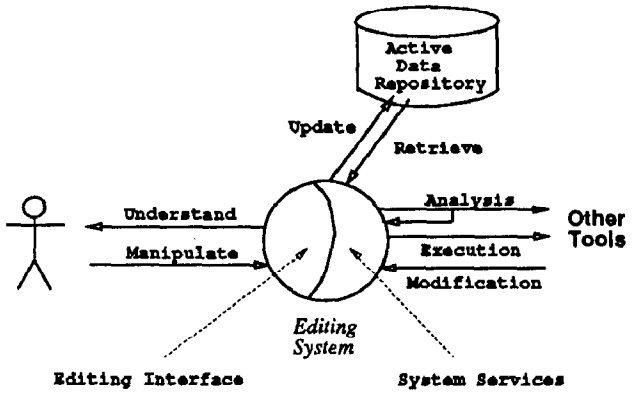

Figure 1: Editing interface and system services in relation to the environment

a language-based system, each of which relies on particular aspects of supported languages, grow with time.

An effective language-based system must provide a *language description* mechanism that defines how the system handles each language. This mechanism must support basic language *definition*, and must extend in many ways. It must be made as accessible and convenient to use as possible, preferably with natural, declarative specifications.

Users are fluent. Users know their primary languages and tools well. An editing interface must deliver services that augment productivity; it must not sacrifice flexibility or power in the name of safety or learnability. For example, experienced developers will not trade away the flexibility of unlimited text editing for the safety of enforced syntactic correctness.

On the other hand, all users sometimes need extra support, for example when confronted with unfamiliar languages. We believe that support for experienced users can be specialized to handle these cases. The converse—supporting the experienced practitioner with tools tuned to the novice—is far more difficult.

2.2 The Role of the System

A language-based editing/browsing system provides the primary interface between people and integrated environments containing the documents they manage. Positioned this way, between users, tools, and documents (Figure 1), the system is uniquely situated to gather and present information *about* documents. In this model, users interact with documents through the editing interface; tools interact with documents through the system services; they communicate with one another via an active data repository. The editing interface and the system services provide alternate projections (views) of the document as well as analysis for the user.

Gathering and presenting information. Software developers acquire and exploit many kinds of information as they examine and modify complex documents [27, 37]. The system can support these activities by gathering and presenting information suitable for the particular task at hand.

For example, language-based systems can check that a document is well-formed. Information derived during these checks can be used to support language-based editing as well as to enable a user to locate and document components that violate constraints of the underlying language. More advanced interaction requires more elaborate analysis. For example, language-based formatting (prettyprinting), traditionally based only on surface syntax, should be sensitive to scopes, types, and def-use relationships, as well as to local conventions and even distinctions such as "mainline" vs. "error handling" code. These kinds of analysis move far beyond simple error checking: they involve knowledge of particular organizations, techniques, and systems. Although this kind of information must be broad in subject domain, it need not be deep (in the sense that program *plans* [38, 48] and *clichés* [46] are deep) to be useful.

When a system derives a considerable amount of information about the document being edited, care has been taken that the system not "do too much" [42]. Often information is most useful if it is not forced on the user but is presented only upon request. An example is the information that a structured document is incomplete.

Maintaining and sharing information. Complex, expensive analyses in such a system are economic 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 performance results and version history.

2.3 The Structure of Documents

Documents represent richly connected, overlapping webs of information having many structural aspects. Each aspect is more relevant for some kinds of users than for others and for some tasks more than for others. A system in the role we envision must support many kinds of users, many tasks, many structural aspects.

Text. Despite arguments to the contrary [59, 61] many language-based editors restrict how and when users may manipulate documents as *text*. Experienced professionals, however, will generally not tolerate re-

strictions on this natural and convenient mode of interaction. At the same time, the full visual power of the textual medium is seldom realized in editing interfaces. The value of high quality typography for natural language documents is well established. Recent studies suggest the same potential benefits for programs [3, 44].

Language-Based Structures. Many relationships among document components (and among documents) can be derived from documents by using descriptions of their underlying formal languages. Examples include 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. Although formal syntax has often been taken to be the primary (and most interesting) decomposition, others are at least as important to users.

Other Structures. But not all structures can be derived using a language description. For instance users of editors like ED3 [52] and the Cedar editor (Tioga) [55] may explicitly specify for each document a hierarchical decomposition that may or may not correlate with structures in the underlying formal language. Outline processors are editors for precisely this kind of structure where the underlying language is simple text.

In some cases a single document may be encoded in more than one language. In this case, the relationships across languages and their appearances within the document become interesting. The programming environment Mentor [20] supports such nesting of languages.

Hyperlinks within and among documents are a third example of document structure that are neither textual nor language-based.

2.4 Project Focus

The design and implementation of *Pan I* addressed three general problems:

- To efficiently provide combined text-oriented and language-oriented editing within a coherent user interface,
- To exploit the assumption of a persistent database in which documents will ultimately reside, and
- To develop language description techniques that support incremental analysis, that are relatively accessible and simple to use, and that can be extended well beyond simple language definition.

Combined solutions to these three problems give the prototype considerable leverage as a research platform for developing more advanced document editing and viewing capabilities, capabilities that are ultimately aimed at enhancing document comprehension and manipulation by users.

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

2.5 Design Strategies

The above observations and goals led to the adoption of a small number of pervasive, interdependent strategies for the design of *Pan I*. The realization of these strategies will be treated in more detail in following sections.

Syntax Recognition. To present the appearance of a "smart text editor," one that also supports language-based interaction, *Pan I* is *syntax-recognizing*, as are Babel [28], the Saga editor [34], and SRE [12]. 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 [53], Mentor [20], and Gandalf [24] *syntax-directed*[3]. As a byproduct 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 document editing demands that derived information be kept current. Many of the analyses envisioned for *Pan* are computationally expensive, so incremental methods are necessary to provide adequate performance during the foreseeable future. Furthermore, non-recoverable information associated with document components (for example direct annotations by users and information imported from other tools) can only be preserved by incremental methods.

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 the traditional term "language errors." This approach acknowledges that experienced users often introduce variances deliberately while working toward a

[3] 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.

desired result; users should not be penalized by the system's failure to understand the process [39].

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, rather than an interface between user and internal representation, language-oriented operations in *Pan* are 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 associated complexity.

Extensibility and Customization. Flexibility at many levels is important for *Pan*'s combined role as tool and research platform. It must adapt conveniently to enormous variations among individual users, among projects (group behavior), and among sites. For use as a research platform the system must be built on a flexible framework designed to accommodate many kinds of variation and evolution [36]. Adding languages by description is one such mechanism, but many others are important too.

Pragmatics. Two final issues can be crucial to the success of a system like *Pan*. Experience shows that it must be acceptably fast; users are seldom willing to compromise on this, even in the name of additional or improved functionality. It must also integrate smoothly into existing, well established working environments. This is an issue both for the user interface, where being perceived as excessively different is a handicap, and for data interchange.

3 System Infrastructure

All services in *Pan* depend on a rich and flexible infrastructure, designed to support experimentation with document analysis techniques and the design of editing interfaces. This section describes a few important aspects of that infrastructure.

The language-based mechanisms described in Section 4 use this infrastructure, as do a few simple services that are not language-based at present: a browsing interface to the file system; a hypertext-like browser for UNIX[4]; and an elaborate internal help and documentation system that is configurable for both developers and end users.

3.1 Basic Editor Services

Pan is, before anything else, a high quality text editor. It supports bit-mapped, multiple-font, mouse-based text editing in windows, in the spirit of Bravo [35] and its many successors. Extension and customization facilities are fully integrated with undo facilities and the help subsystem.

Editing in *Pan* is fundamentally textual. The same editing services are provided in every *buffer*, whether or not the document being edited in that buffer is written in a language that the editor is prepared (by prior language description) to analyze. *Pan*'s user interface does not isolate it from other tools in the environment, in contrast to many syntax-directed editors. Users familiar with Emacs [49] find the transition between the two editors smoothed by compatible key bindings [8].

3.2 Viewing

A buffer's conceptual text presentation may appear in any number of *windows*, each updated incrementally as text changes. All windows on a buffer share a single, visible *selection* that appears as underscored text in any window in which it happens to be visible. The selection persists independently of cursor movement. Each window has its own scroll position and *cursor* that persist, even when the window is not visible.

Each *Pan* window has an associated *panel* that contains an identifying title, an annunciator for messages, and various mouse-activated controls. The panel also displays a configurable collection of *panel flags*, single-character glyphs that appear and disappear to reveal important internal state to the user (for example, whether the contents of the buffer have been modified).

Each character in a buffer has an associated font code, an index into a user-configurable *font map* that may contain fonts of differing heights and widths. Additional facilities are available for superimposing more information upon the textual display: underlining, stipple patterns, colored inks, and color background shading (in the manner of "highlighter" pens). Background shading may be selected independently to ink color.

3.3 Extension Language

Although some of these *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 special extension language. Although that access can be abused, it has not proven to be a problem in practice.

Variables. *Variables* are *scoped* dynamically by buffer instance, buffer class, and a global scope. Scoping is used for implementing buffer-specific services, for organizing buffer class services such as the command bindings common to all documents sharing a common base language, and so on. Many other facilities in the

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

infrastructure are built upon *Pan* variables, including user-settable options, keyboard bindings, panel flags and font maps. *Pan* variables may be made "active" by the dynamic addition of *notifiers* to be called when values change.

Function and Macro Definition. A general function and macro definition facility provides automatic integration with the internal documentation system, apropos keywords for the help system, and generic undo. Functions can be called from other functions or can be bound to keystroke sequences or menu selections. Arguments for bindable commands are collected automatically from the user by type-specific prompters.

Generic Exception Handling. *Pan* distinguishes three categories of internal exceptions: announcements, warnings, and errors. Each corresponds to a different policy for displaying information to the user and unwinding (resetting) the run-time command dispatcher. To provide context-sensitive behavior [57], the exception handlers may be rebound dynamically. Authors of simple extension code may ignore exception handling without fear of serious system breakage, greatly simplifying the construction of new services.

4 Document Analysis

Document analysis in *Pan* relies on two components: *Ladle*[5] [13], and *Colander*[6] [5]. *Ladle* manages incremental lexical and syntactic analysis; it includes both an offline 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 offline preprocessor and a run-time component. The editing interface [8] 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 Descriptions

A *Pan language description* contains declarative information for use by each of *Pan*'s three components[7].

- Lexical and syntactic information, used by *Ladle*, describes the syntax of the language and defines an internal tree-structured representation (Section 4.2).
- 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.
- User interface specifications configure the editing interface for the language (Section 6.1).

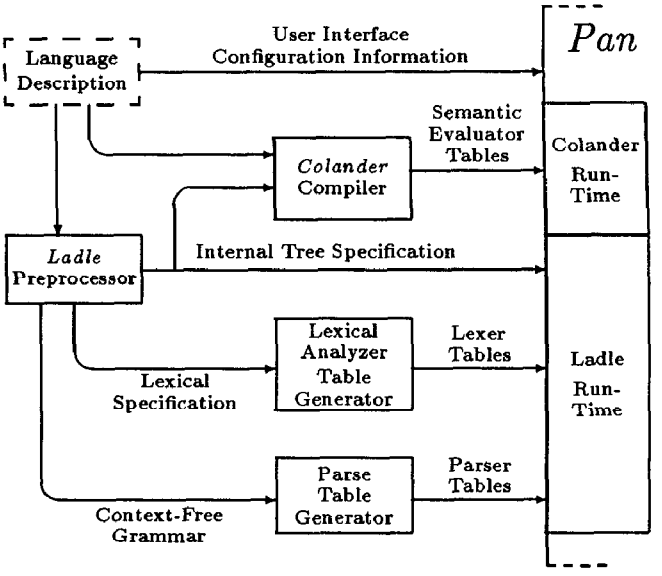

Figure 2: Language Description Processing

Figure 2 illustrates the flow of information from a language description to the run-time *Pan* system, for either preloading or dynamic loading at run time.

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 [22], for example the well known "typedef" problem in C. Research into general solutions to these problems within *Pan* is currently underway at UC, Berkeley.

Multiple *Pan* language descriptions can be written for a single language, suited for different users and different tasks. One language description might reveal to the user only a single derived structure of a document, for example the call graph of a program, and hide the rest. A second language description for the same base language might perform full contextual-constraint checking and permit full access to the internal tree. When there is no sharing of internal structures, alternate descriptions can be independent; otherwise the *Ladle* portion of the

[5] **Language Description Language**

[6] **Constraint Language and Interpreter**

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

descriptions must be shared as well. A second area of active research concerns the *layering* of language descriptions so that multiple views of a single abstract syntax can be managed effectively.

4.2 Ladle

An *abstract syntax* is described to *Ladle* by 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 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 regular expressions as well as 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 *Ladle*'s syntactic error recovery mechanisms (invoked during parsing), with important implications for 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*[8] [6], a relation ensuring that:

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. Both grammars describe "almost" the same formal language, subject to the renaming or erasing of symbols.
2. Efficient incremental transformations from concrete to abstract and from abstract to concrete can be 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.

The *Ladle* preprocessor generates the tables needed to describe the internal tree representation as well auxiliary tables needed during incremental parsing and error recovery. Standard lexical analyzer generators and LALR(1) parser generators are also invoked, as shown in Figure 2.

To date, syntactic descriptions have been written for Modula-2, Pascal, Ada, *Colander*, and for *Ladle*'s own language description language. Descriptions are being developed for a variety of other languages, including C, C++, and FIDIL [26].

4.3 Colander

Colander supports the description and incremental checking of contextual constraints. Constraints include non-structural 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, non-local linkages within or among documents.

Each of the well known specification formalisms for contextual constraints has drawbacks. Attribute grammars [19, 45] usually specify only attributes and their interrelationships; a separate formalism—usually a general purpose programming language—must be used to specify semantic functions and data types. In practice much of the interesting information in an attribute grammar resides in auxiliary code. Moreover, there is no easy way to make information in the attribute values available to other tools [29]. Action routines [33, 41] require one to deal with all aspects of incrementality explicitly. Context relations [4], like attribute grammars, relegate many of their description details, such as name resolution rules, to a separate formalism.

Our approach is based on the notion of *logical constraint grammars*. Like the other approaches, a context-

[8] Butcher has recently recast this work in terms of *grammatical expansion* [13]; *Ladle* will be updated accordingly.

free grammar used as a base. Contextual constraints are expressed using a logic programming language[9]. 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 LCGs include the kinds of analysis performed by tools such as Masterscope [40] 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[10] as well as the run-time tables required for consistency maintenance. The consistency manager, a simple reason maintenance system [21, 47], invokes the evaluator to (re)attempt a goal. The evaluator, in turn, collects the information maintained by the consistency manager.

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.

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 [30].

The incremental parser maintains a summary of changes for use by *Colander*'s incremental constraint checker (Section 5.5), as well as by any other services that must track document changes.

4.5 Information Retention

Since subtrees may be heavily annotated (both internally 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 each changed area and the tree root. Widely-shared data often appears closer 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 node reuse. The parser keeps a stack of "divided" tree nodes and when new nodes are needed they are taken from this stack if possible. A node can be 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.

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

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 treats such problems as "variances," not as errors, and make every effort to treat them as relatively normal occurrences. In particular, the editing interface imposes no special restriction on users in the presence of variances, but makes the diagnostics available upon request. *Pan*'s internal document representations are automatically extended to admit variances and to retain as much information as possible in their presence.

Whenever 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. This is the only case in which the analysis of a document can "fail"; in all other cases, recovery mechanisms are automatically invoked.

During parsing *Pan* uses a simple, effective, panic mode mechanism [18] for syntactic error recovery. 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 [34]. Any extant annotations on the subtrees within the error subtree are preserved, including annotations created by *Colander*. When the user corrects the variance, prior annotations

[9] Logical constraint grammars should not be confused with *constraint logic programming* [16] a generalization of logic programming.

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

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 subtrees within error subtrees are simply ignored. Unsatisfied contextual constraints are another kind of variance, resulting in an annotation on the offending node.

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 [5, 7] 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 can 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 logic database. This contrasts with the approach of Horwitz and Teitelbaum [29], in which the relational database model had to be greatly enriched in order to support a coupling with attribute grammars.

The extensibility of this approach, above and 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.

5.1 Definitions

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 a 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.

5.2 Incremental Evaluation

Inconsistencies between documents and their derived information arising from incremental changes are detected by a consistency manager, which is a simple form of 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. The process continues until consistency is restored.

Careful selection of goals to be retried after database updates, either additions or deletions, is crucial to efficient incremental evaluation. Removal of data from the database, the simpler of the two cases, is handled using dependency-directed backtracking [50]. 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* 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 upon 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 a 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.

For a given LCG description, satisfying some goals requires satisfying other goals. For instance, one goal may assert a datum that will be used by another. Like unrestricted attribute grammars, circularities can arise. 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. The 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.

5.3 LCGs and Logic Programming

Making LCGs practical required several modifications to the basic Prolog model of logic programming [51].

Partitioned Database. The logic database accessed and modified by the goals is explicitly structured into *collections* of *tuples* (data values). Both collections and tuples are first class objects: they can be created and destroyed dynamically. Collections are created and tuples are added to collections as a side effect of satisfying a goal. Tuples can contain references to collections, but cannot contain unbound 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 instance, 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 requirement simplifies the preprocessing needed by the consistency manager, and also improves run-time performance.

Ownership. Every collection and tuple in the database is *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.

Separating the context from the goals in a logical constraint grammar 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. 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 [43].

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

5.4 Example

Figure 3 shows a very simple LCG for a language that requires that each name be defined before it is used. This notation resembles the *Colander* language. A name prefixed by "$" denotes a node in the abstract syntax tree; "$$" denotes the node associated with the goal currently being satisfied. The notation < *?Form*, *?collection* > indicates that *?Form* is to be evaluated with respect to the given collection of tuples. An "entity" is just a unique marker that can be used to represent linguistic objects such as variables. The procedure "`lookup`" specifies how scopes (represented here using contexts directly) are searched when resolving a name. In this example, "`lookup`" implements nested block structure. The constraint associated with ⟨*use*⟩ will look for a binding created by a ⟨*def*⟩ or an ⟨*import*⟩. In a complete description, a goal associated with the definition of a procedure would create a new context to be inherited by its children; this context would include a new scope as accessed by "`lookup`."

5.5 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 or the usability of the description language, or both.

Pass-Structured Evaluation. *Colander* partitions the goals associated with each syntactic structure into two classes: those 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.

Multiple Kinds of Collections. *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 target language. Typically entities are 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, so a collection may not contain more than one property value with a given property name. Distinguishing between facts and

Facts:

```
declared(?name, ?entity)          /* ?Name bound to ?entity */
imported(?name, ?entity)          /* ?Name imported as ?Entity */
enclosing-scope(?scope)           /* Fact representing the next outer scope */
type-of(?entity, ?type-mark)      /* Type of ?entity — either "id" or "proc" */
```

Primitive Procedures:

```
assert(?fact)                     /* add ?fact to global collection */
assert(?fact, ?Collection)        /* add ?fact to ?collection */
context(?loc, ?Scope)             /* bind ?Scope to single context of ?loc */
new-entity(?entity)               /* bind ?entity to unique marker */
string-name(?loc, ?name)          /* bind ?name to string name of ?Loc */
```

User-Defined 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>.
```

Grammar:

```
⟨document⟩ → ⟨def⟩* ⟨use⟩*
⟨def⟩      → "DEF" id
             :- context($$, ?Scope), string-name($id, ?name),
                not(<visible(?name, ?Dtemp), ?Scope>), new-entity(?entity),
                assert(<declared(?name, ?entity), ?Scope>), assert(type-of(?entity, "id")).
⟨def⟩      → "IMPORT" id ...                                   /* Similar to a DEF */
⟨def⟩      → "PROC" id ⟨def⟩* ⟨use⟩* ...                       /* Create new context to be inherited by children */
⟨use⟩      → "USE" id
             :- context($$, ?Scope), string-name($id, ?name),
                <lookup(visible(?name, ?entity)), ?Scope>, type-of(?entity, "id").
⟨use⟩      → "CALL" id ...                                     /* Check that id is declared as a procedure */
```

Figure 3: Fragment of Simple Logical Constraint Grammar

properties can improve the performance of the incremental evaluator.

Subtrees can also be considered as collections holding subtree properties. A *subtree property* is a named value associated with a subtree. Structural information about the internal tree is represented using subtree properties maintained by the system. Subtrees are created and destroyed by *Ladle* during syntactic analysis.

Maintained Subtree Properties. The value of a *maintained subtree property* is defined by a local procedure associated with that subtree. Maintained properties are evaluated on a demand basis. *Colander* memoizes the values computed by maintained properties in order to minimize recomputation.

Maintained properties are similar to attributes in an attribute grammar, and the procedures that define the values of maintained properties play a role similar to attribute functions. The flow of property values in *Colander* trees is usually much simpler than in attribute grammars. Inherited values typically propagate via the context datapool, and most values will reside in the database. While it is possible to emulate an attribute grammar directly, it is far more efficient to use the database and the context for moving values through the tree.

This restriction is not as severe as it might appear. In most attribute grammar descriptions, inherited attributes either summarize relatively local structural information about the internal tree or else they consist of a "symbol table" containing non-local information. *Colander* subsumes the "symbol table" into the database along with other information about the expression. Local structural information can be passed like inherited attributes by creating and propagating new context datapools containing that information. Synthesized values propagate from leaves towards the root as in attribute grammars; these appear frequently in *Colander* descriptions.

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.

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 retained for possible display to the user.

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 can be used to calculate all solutions to a goal. It is reevaluated whenever the set of solutions might have changed. The function notever is a form of negation that is monitored by the consistency manager. The results of the notever primitive are under consistency maintenance. If a new solution arises that would cause the not to fail, then the goal containing the notever will be retried.

6 User Services

The ultimate purpose of *Pan* is to assist its intended users. This section discusses some of the technical problems associated with providing user services. A more thorough treatment of *Pan*'s approach to delivering language-based technology to users appears elsewhere [58].

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 representations is driven primarily by implementation issues associated with clients of the data. Experience with *Pan* reveals a strong influence on the abstract grammar by the *Colander* specification. These influences are unrelated to the way users understand document structure.

Pan decouples internal representation from what the user sees. The language description mechanism provides a loose framework in which the author of each language description is expected to design a model. This framework is based on two assumptions about how people understand syntactic 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*, the basis for structure-oriented selection, navigation, highlighting, and editing. Operand classes are arbitrary, possibly overlapping[11] collections of document components. Operand class membership is derived dynamically, being the results of a query against the database of derived information.

Pan supports several varieties of operand classes:

- Operand classes "Character", "Word", and "Line" are defined for all textual documents.
- Operand classes defined in each language description specify the structural components of the language that will be revealed to the user. For example, our standard Modula-2 description includes classes named "Expression", "Statement", "Declaration", and "Procedure".
- Operand classes "Lexeme", "Syntactic Error" and "Unsatisfied Constraint" are available for all documents having an underlying language description. The classes "Syntactic Error"[12] and "Unsatisfied Constraint" denote the sets of syntactic and contextual-constraint variances, respectively. Overlapping classes are put to good use here, allowing the user to treat such a component in either of two ways; for example the structure representing a malformed statement might be in both the "Statement" and "Syntax Error" classes.
- The operand class "Query Result" allows a user to identify structures based on his or her own query.

Diagnostics. Parts of each language description specify diagnostic messages that are to be generated when document analysis reveals particular variances. The presence of variances precipitates no special action, other than the appearance of designated panel flags. At any time, however, the the user may invoke one of the many services that helps make diagnostics visible.

[11] The relationship between tree operators and operand classes is many-to-many, in contrast to the related notion of operator-phyla [32]. The operand classes for a language need not be complete; structures not in any class are essentially invisible to the user.

[12] Although "syntax errors" are just one kind of variance in the *Pan* system, to users they go by their traditional name.

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. All this reinforces the illusion that an ill-formed document is not much different than a well-formed document.

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 for a particular document. Rather than hide the fact completely, *Pan*'s default configuration adds a special panel flag that appears when this is the case. Derived information is exploited and possibly made visible only through a number of specific services, all optional and under user control. This section describes the implementation of a few such services, deferring discussion of language-based editing to Section 6.4.

Presentation Enhancements. Several visual enhancements help draw attention to particular document components. *Font shifts* reveal the lexical category of text, for example language keywords, identifiers, and comments. In our experience font shifts contribute significantly to program readability, but only when tuned for each language using *Pan*'s font maps.

Prettyprinting reindents documents to reveal syntactic structure. More advanced forms of prettyprinting for program documents, including semantically-driven elision, are under development [10].

Structural highlighting allows text associated with specified operand classes to be rendered with one of several (generally independent) special effects: background color, stipple patterns, and ink color. For example, all of the structures designated by a particular query might be rendered using blue ink, or a user may request continual highlighting of all text associated with variances. Although simply highlighting variances doesn't reveal as much information as diagnostic messages, experienced programmers can often diagnose simple variances at a glance, once attention is drawn to them.

The Operand Level. Each *Pan* window has a current *operand level*, which the user selects from a language-specific menu of operand classes. The operand level is a very weak input mode that modulates the operation of five generic commands: Next, Previous, Select, Extend Selection, and Delete. The operand level affects no other commands, and a user may choose not to use the level-sensitive versions at all. In particular, the operand level neither inhibits nor modulates text-oriented editing at any time.

Structural Navigation. The operand level enables specialized, language-specific forms of navigation. When the operand level is "Statement", for example, the user may press the left mouse button anywhere to select the "nearest" (based on a heuristic) syntactic component that meets the definition of that operand class. Commands Next and Previous perform a tree walk, selecting only subtrees of the appropriate operand class.

Structural navigation also enables examination of diagnostic messages. Any structural selection that occurs at one of the special operand classes "Syntax Error" or "Unsatisfied Constraint" causes a diagnostic message associated with the variance to appear in the window's annunciator. A user who sees the flag indicating the presence of variances can simply walk through all of them, viewing the location and diagnostic of each one 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 language-based information may be derived from text, is no exception. During text-oriented editing derived information maintained by the system will sometimes disagree with what the user sees. For example lexical font shifts would be incorrect immediately after the textual transformation of a statement into a comment.

A special panel flag appears when text and derived data are inconsistent. Continuously visible enhancements like font shifts usually remain *almost* correct in ways easily understood by experienced users.

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 periods of inconsistency, *Pan* triggers reanalysis before attempting it. Since analysis (almost) always succeeds, this policy does not restrict the user, although it may cause delay.

Pan's reanalysis policy is a lazy one, based upon the assumption that the user understands the general state of the document and can judge the tradeoffs involved. Incremental analysis is only performed when requested by the user, either *implicitly* by invoking an operation that triggers automatic reanalysis or *explicitly* by invoking the command Analyze-Changes. Nothing prevents a *Pan* user from typing an entire document without once invoking analysis. The *Pan* approach is to encourage frequent analysis by making it cost-effective to the user.

6.4 Mixed-Mode Editing

Pan's fundamental approach to language-based editing is to broaden the user's options, not narrow them. The user should be able edit textually any time, any place in

the document presentation; it should be equally possible to edit in terms of derived information any time, any place.

A Dual Aspect Cursor. Any *Pan* editing command, 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 before performing any operations that require it. The second mechanism 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. At present, the cursor's structural location is revealed by turning the component's textual presentation into the current text selection (displayed by underlining). Any operation that sets the structural cursor also positions the text cursor at the first character in the structure's textual presentation.

Any editing operation that requires a cursor location simply 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 vs. extended cursor" problem [54] by providing both behaviors simultaneously.

Simple Editing. The prototype implementation supports no user commands that modify internal document structure directly. The Delete command invoked with a structural selection, for example at operand level "Statement", achieves the same effect by removing the text associated with the statement. The internal representation of the deleted component remains in the tree until the next reanalysis, but it is invisible to the user because automatic reanalysis will delete it before any commands can use it. The Cut command simply places text in the clipboard.

The command Paste simply inserts text from the clipboard. If the context is appropriate, subsequent incremental analysis derives the equivalent structural information quickly.

This implementation costs a small amount of analysis time by discarding derived information when the user moves structural components. On the other hand, it guarantees the integrity and well-formedness of the document's internal representation, since the language definition is already built into *Pan*'s parser.

Complex mechanisms for direct structural editing can be a source of confusion to the user, since those editing operations may fail. Worse, they may fail for the kinds of reasons we attempt to hide. For example, it seems reasonable to copy the list of identifiers appearing in the formal parameter list of a procedure definition and paste it into a call to that procedure. Although the two lists of identifiers might appear identical and be closely related conceptually, there may be sound implementation reasons for different internal representations in the two contexts. We prefer to avoid strategies that involve guessing the user's intent.

When a structurally inspired Cut and Paste sequence in *Pan* violates the underlying language definition, the operations succeed anyway and the problem is diagnosed by precisely the same mechanisms that handles other variances.

The cost at present of this text-based implementation is the loss of any non-derivable annotations on document components during Cut and Paste sequences. We have developed, but not yet added, a strategy that avoids this information loss and provides functionality that is equivalent to direct structural operations.

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. The best developed collection of these services in *Pan* at present deals with the location and diagnosis of variances, as described above. This section describes a few other examples that either have been prototyped or will be soon.

One of the few forms of query supported by ordinary text editors is textual search. Searching in *Pan* can draw on *any* information in the repository, as exemplified by the structural query mechanism described earlier. For example, one query-based command allows the user to point at a variable name and then ask to see the declaration and all uses of that variable. A closely related command allows the user to point at a variable and move the cursor to the corresponding declaration; this is only one example of a command that follows hypertext-like links defined by the underlying language.

Like all full-functioned text editors, *Pan* also supports textual replacement based on regular expression matching. However, one often intends that the replacement depend on the language structure, not on the textual structure, even when the two are similar. For example, word replacement in natural language documents is tricky using regular expressions. One wants to avoid replacing occurrences embedded in other words, so a simple specification of the search string does not suffice; at the same time, it is difficult to describe all characters (including beginning and end of line) that might mark word boundaries. The answer involves specific commands that use *Pan*'s derived information to replace only whole components. Even more powerful versions replace only those occurrences of a name that are related according to the rules of the language, for example when renaming a variable in a program.

7 Retrospective

Pan I 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. Related issues, including support for novice programmers and learning environments, support for program execution, graphical display and editing, and the actual design and implementation of a persistent software development database, were deferred. The *Ensemble* project, which will create a successor to *Pan I*, is addressing some of them.

Ongoing research projects are using the leverage gained from *Pan I*. Projects near completion include a study of prettyprinting using *Pan* and *Colander* [10]; the development of advanced document analysis techniques using *Colander* to specify and control user centered program viewing [56]; the development of new language descriptions; and investigations into ways to strengthen *Pan*'s language description techniques [22].

With the exception of a simple non-editable 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 [14, 15].
2. The extension of editing and viewing to a wide range of media—text, graphics, sound, and video.
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. However, complete descriptions of programming languages 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 [31] as used in Centaur[13] [11].

8 Acknowledgements

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.

[13] The Typol descriptions used in Centaur are already compiled to Prolog; adapting their compilation techniques to LCG's, and introducing the side effects implied by the LCG database in a controlled way seems straightforward.

References

[1] ACM. *Proc. of the ACM SIGPLAN SIGOA Symposium on Text Manipulation* (Portland, Oregon, June 8–10, 1981). Appeared as Sigplan Notices, 16(6), June 1981.

[2] AMBRAS, J., AND O'DAY, V. Microscope: A knowledge-based programming environment. *IEEE Software 5*, 3 (May 1988), 50–58.

[3] BAECKER, R. M., AND MARCUS, A. *Human Factors and Typography for More Readable Programs.* ACM Press, New York, New York, 1990.

[4] BAHLKE, R., AND SNELTING, G. The PSG system: From formal language definitions to interactive programming environments. *ACM Trans. on Programming Languages and Systems 8*, 4 (1986), 547–576.

[5] BALLANCE, R. A. *Syntactic and Semantic Checking in Language-Based Editing Systems.* Ph.D. dissertation, Computer Science Division—EECS, University of California, Berkeley, California, 94720, Dec. 1989. Available as Technical Report No. UCB/CSD 89/548.

[6] BALLANCE, R. A., BUTCHER, J., AND GRAHAM, S. L. Grammatical abstraction and incremental syntax analysis in a language-based editor. In *Proc. of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 22–24, 1988), ACM, pp. 185–198. Appeared as Sigplan Notices, 23(7), July 1988.

[7] BALLANCE, R. A., AND GRAHAM, S. L. Consistency maintenance for contextual constraints. In preparation., July 1990.

[8] BALLANCE, R. A., AND VAN DE VANTER, M. L. Pan I: An introduction for users. Technical Report No. UCB/CSD 88/410, Computer Science Division—EECS, University of California, Berkeley, California, 94720, Mar. 1988.

[9] BALLANCE, R. A., VAN DE VANTER, M. L., AND GRAHAM, S. L. The architecture of Pan I. Technical Report No. UCB/CSD 88/409, Computer Science Division—EECS, University of California, Berkeley, California, 94720, Mar. 1988.

[10] BLACK, C. L. PPP: The Pan program presenter. Technical Report No. UCB/CSD 90/589, Computer Science Division—EECS, University of California, Berkeley, California, 94720, Sept. 1990.

[11] BORRAS, P., CLÉMENT, D., DESPEYROUX, T., INCERPI, J., KAHN, G., LANG, B., AND PASCUAL, V. CENTAUR: the system. In Henderson [25], pp. 14–24.

[12] BUDINSKY, F. J., HOLT, R. C., AND ZAKY, S. G. SRE—a syntax-recognizing editor. *Software—Practice & Experience 15*, 5 (May 1985), 489–497.

[13] BUTCHER, J. Ladle. Master's thesis, Computer Science Division—EECS, University of California, Berkeley, California, 94720, Nov. 1989. Available as Technical Report No. UCB/CSD 89/519.

[14] CHEN, P., COKER, J., HARRISON, M. A., MCCARRELL, J., AND PROCTER, S. The VorTEX Document Preparation Environment. In *Proc. of the 2nd European Conference on TEX for Scientific Documentation* (Strasbourg, France, June 19–21 1986), J. Desarménien, Ed., no. 236 in LNCS, Springer-Verlag, pp. 45–54.

[15] CHEN, P., AND HARRISON, M. A. Multiple representation document development. *IEEE Computer 21*, 1 (Jan. 1988), 15–31.

[16] COHEN, J. Constraint logic programming languages. *Communications of the ACM 33*, 7 (July 1990), 52–68.

[17] CONRADI, R., DIDRIKSEN, T. M., AND WANVIK, D., Eds. *Advanced Programming Environments* (Berlin, Heidelberg, New York, 1986), no. 244 in Lecture Notes in Computer Science, Springer-Verlag.

[18] CORBETT, R. P. *Static Semantics and Compiler Error Recovery.* Ph.D. dissertation, Computer Science Division—EECS, University of California, Berkeley, California, 94720, June 1985. Available as Technical Report No. UCB/CSD 85/251.

[19] DERANSART, P., JOURDAN, M., AND LORHO, B. *Attribute Grammars: Definitions, Systems, and Bibliography.* No. 323 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, New York, 1988.

[20] DONZEAU-GOUGE, V., HUET, G., KAHN, G., AND LANG, B. Programming environments based on structure editors: The Mentor experience. Rapports de Recherche 26, INRIA, June 1980.

[21] DOYLE, J. A truth maintenance system. In *Readings in Artificial Intelligence*, B. L. Webber and N. J. Nilsson, Eds. Tioga, Palo Alto, California, 1981, pp. 496–516.

[22] FORSTALL, B. T. Experience with language description mechanisms in Pan. PIPER Working Paper 90-3, Computer Science Division—EECS, University of California, Berkeley, California, 94720, 1990.

[23] GOLDBERG, A. Programmer as reader. *IEEE Software 4*, 5 (Sept. 1987), 62–70.

[24] HABERMANN, A. N., AND NOTKIN, D. Gandalf: Software development environments. *IEEE Trans. on Software Engineering SE-12*, 12 (Dec. 1986), 1117–1127.

[25] HENDERSON, P., Ed. *ACM SIGSOFT '88: Third Symposium on Software Development Environments* (1988).

[26] HILFINGER, P. N., AND COLELLA, P. Fidil: A language for scientific programming. In *Symbolic Computation: Applications to Scientific Computing*, R. Grossman, Ed. SIAM, 1989, pp. 97–138.

[27] 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 Publishing, Norwood, NJ, 1987, p. 33.

[28] HORTON, M. R. *Design of a multi-language editor with static error detection capabilities.* Ph.D. dissertation, Computer Science Division—EECS, University of California, Berkeley, California, 94720, 1981.

[29] HORWITZ, S., AND TEITELBAUM, T. Generating editing environments based on relations and attributes. *ACM Trans. on Programming Languages and Systems 8*, 4 (Oct. 1986), 577–608.

[30] JALILI, F., AND GALLIER, J. H. Building friendly parsers. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, January 25–27, 1982), ACM, pp. 196–206.

[31] KAHN, G. Natural semantics. Tech. Rep. 601, INRIA, Feb. 1987.

[32] KAHN, G., LANG, B., MÉLÈSE, B., AND MORCOS, E. Metal: A formalism to specify formalisms. *Science of Computer Programming 3* (1983), 151–188.

[33] KAISER, G. E. *Semantics for Structure Editing Environments.* Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, Pennsylvania, 15213, May 1985.

[34] KIRSLIS, P. A. C. *The SAGA Editor: A Language-Oriented Editor Based on Incremental LR(1) Parser.* Ph.D. dissertation, University of Illinois at Urbana-Champaign, Dec. 1985.

[35] LAMPSON, B. W. *Bravo Users Manual.* Palo Alto, 1978.

[36] LANG, B. On the usefulness of syntax directed editors. In Conradi et al. [17], pp. 47–51.

[37] LETOVSKY, S. Cognitive processes in program comprehension. In *Empirical Studies of Programmers*, E. Soloway and S. Iyengar, Eds. Ablex Publishing, Norwood, NJ, 1986, pp. 58–79.

[38] LETOVSKY, S., AND SOLOWAY, E. Delocalized plans and program comprehension. *IEEE Software 3*, 3 (May 1986), 41–49.

[39] 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. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, pp. 411–432.

[40] MASINTER, L. M. Global program analysis in an interactive environment. Tech. Rep. SSL-80-1, Xerox Palo Alto Research Center, Palo Alto, 1980.

[41] MEDINA-MORA, R., AND FEILER, P. H. An incremental programming environment. *IEEE Trans. on Software Engineering SE-7*, 5 (1981), 472–481.

[42] NEAL, L. R. Cognition-sensitive design and user modelling for syntax-directed editors. In *CHI+GI* (1987), pp. 99–102.

[43] NORD, R. L., AND PFENNING, F. The Ergo attribute system. In Henderson [25], pp. 110–120.

[44] OMAN, P., AND COOK, C. R. Typographic style is more than cosmetic. *Communications of the ACM 33*, 5 (May 1990), 506–520.

[45] REPS, T., TEITELBAUM, T., AND DEMERS, A. Incremental context dependent analysis for language based editors. *ACM Trans. on Programming Languages and Systems 5*, 3 (July 1983), 449–477.

[46] RICH, C., AND WATERS, R. C. The Programmers Apprentice: A research overview. *IEEE Computer 21*, 11 (Nov. 1988), 10–25.

[47] SMITH, B., AND KELLEHER, G., Eds. *Reason Maintenance Systems and Their Applications.* Series in Artificial Intelligence. Ellis Norwood Limited, Chichester, 1988.

[48] SOLOWAY, E., AND EHRLICH, K. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering SE-10*, 5 (Sept. 1984), 595–609.

[49] STALLMAN, R. M. EMACS: The extensible, customizable, self-documenting display editor. In *Proc. of the ACM SIGPLAN SIGOA Symposium on Text Manipulation* [1], pp. 147–156. Appeared as Sigplan Notices, 16(6), June 1981.

[50] STEELE JR., G. L., AND SUSSMAN, G. J. Constraints. AI Memo No. 502, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, Nov. 1978.

[51] STERLING, L., AND SHAPIRO, E. *The Art of Prolog: Advanced Programming Techniques.* The MIT Press, Cambridge, Massachusetts and London, England, 1986.

[52] STRÖMFORS, O. Editing large programs using a structure-oriented text editor. In Conradi et al. [17], pp. 39–46.

[53] TEITELBAUM, T., AND REPS, T. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM 24*, 9 (Sept. 1981), 563–573.

[54] TEITELBAUM, T., REPS, T., AND HORWITZ, S. The why and wherefore of the Cornell program synthesizer. *SIGPLAN Notices 16*, 6 (June 1981).

[55] TEITELMAN, W. A tour through Cedar. *IEEE Trans. on Software Engineering SE-11*, 3 (Mar. 1985).

[56] VAN DE VANTER, M. L. User-centered program viewing. Research Proposal, Computer Science Division—EECS, University of California, Berkeley, California, 94720, Nov. 1987.

[57] VAN DE VANTER, M. L. Error management and debugging in Pan I. Technical Report No. UCB/CSD 89/554, Computer Science Division—EECS, University of California, Berkeley, California, 94720, Dec. 1989.

[58] VAN DE VANTER, M. L., BALLANCE, R. A., AND GRAHAM, S. L. Coherent user interfaces for language-based editing systems. Technical Report No. UCB/CSD 90/591, Computer Science Division—EECS, University of California, Berkeley, California, 94720, July 1990.

[59] WATERS, R. C. Program editors should not abandon text oriented commands. *SIGPLAN Notices 17*, 7 (1982), 39–46.

[60] WINOGRAD, T. Beyond programming languages. *Communications of the ACM 22*, 7 (July 1979), 391–401.

[61] WOOD, S. R. Z—the 95% program editor. In *Proc. of the ACM SIGPLAN SIGOA Symposium on Text Manipulation* [1], pp. 1–7. Appeared as Sigplan Notices, 16(6), June 1981.