User Interaction in Language-Based Editing Systems

Michael Lee Van De Vanter

Computer Science Division—EECS
University of California, Berkeley
Berkeley, California 94720

December 1992

Dissertation submitted in partial satisfaction
of the requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate Division of
the University of California at Berkeley

Report No. UCB/CSD-93-726

This research was sponsored in part by the Defense Advanced Research Projects Agency (DoD), monitored by the Space and Naval Warfare Systems Command under Contract N00039-88-C-0292, by the Defense Advanced Research Projects Agency under Grant MDA972-92-J-1028, by IBM under IBM Research Contract No. 564516, by a gift from Apple Computer, Inc., a State of California MICRO Fellowship, and an IBM Fellowship. The content of the information does not necessarily reflect the position or the policy of the U.S. Government.

User Interaction in Language-Based Editing Systems

Michael Lee Van De Vanter

Abstract

Language-based editing systems allow users to create, browse, and modify structured documents (programs in particular) in terms of the formal languages in which they are written. Many such systems have been built, but despite steady refinement of the supporting technology few programmers use them today. In this dissertation it is argued that realizing the potential of these systems demands a *user-centered* approach to their design and construction. *Pan*, a fully-implemented experimental language-based editing and browsing system, demonstrates the viability of the approach.

Careful consideration of the intended user population, drawing on evidence from psychological studies of programmers, from current software engineering practice, and from experience with earlier systems, motivates *Pan*'s design. Important aspects of that design include functional requirements, metaphors that capture the feel of the system from the perspective of users, and an architectural framework for implementation.

Unlike many earlier systems, *Pan*'s design hides the complexity of language-based technology behind a set of simple and appropriate conceptual models — models of the system and of the documents being viewed. Responding to the true bottleneck in software production, *Pan*'s services are designed to help users understand software rather than save keystrokes writing it. Furthermore, *Pan*'s design framework provides services that degrade gracefully in the presence of malformed documents, incomplete documents, and inconsistent information.

This research has yielded new insight into the design problem at all levels: the suitability of current language-based technology for interactive, user-centered applications; appropriate kernel mechanisms for building coherent user services; new conceptual models of editing that blend textual and structural operations without undue complexity; and the crucial role of local, site-specific design in the delivery of language-based editing services.

To the memory of my mother

Helen Carol Van De Vanter

Acknowledgements

Pan is the work of many people. Special thanks go to Robert Ballance and Jacob Butcher, from whose collaboration many of *Pan*'s good ideas emerged and were realized. They also built *Colander* and *Ladle* respectively, the two crucial parts of *Pan*'s language-based infrastructure. Important contributions also came from Christina Black, Laura Downs, Bruce Forstall, Mark Hastings, Darrin Lane, William Maddox, John Pasalis, Tim Wagner, and Robert Wahbe.

Yuval Peduel's efforts at keeping my thinking and writing clear have been a great help, as have the support and encouragement of colleagues Gordon Hamachi, Peter Kessler, Paul Kube, Jock MacKinlay, Vicki O'Day, Eduardo Pelegrí-Llopart, Richard Probst, Dain Samples, and Polle Zellweger.

I would like to thank my committee members, Susan Graham, Michael Harrison, and Andy diSessa, for their guidance. My research advisor, Susan Graham, deserves special thanks for her wisdom and foresight throughout the project.

Finally, I would like to thank my wife Karen for her patient support, and my children Rebecca, Garth, and Justin for being themselves.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems are unique among engineered artifacts: to commit a detailed design to a suitable formal notation is to construct the system being designed. The natural preoccupation with notation in software engineering motivates a corresponding interest in tools that help people create and manage increasingly complex software documents.

Language-based editing systems represent an important evolutionary thread in the development of tools for the construction of software. These systems allow users to create, browse, and modify software documents in terms of the formal languages and notations in which they are written (for examples in terms of “statements,” “integer expressions,” and “assignments with type-compatibility problems”) not just in terms of their superficially textual characteristics. However a lack of widespread acceptance has proven something of a disappointment to those who envision the potential contribution of these systems.

One part of the problem has been a lack of language-based technology appropriate for interactive use, in contrast to the much better understood world of batch-oriented program compilation. Several generations of experimental language-based editing systems have made significant progress with the technology [7,17,29,94,104], and practical systems of this kind are now within reach, for example the single-language SMARTSystem[1] for C [97].

Growing experience with these almost-practical systems suggests that more attention now be given to another part of the problem: user interaction. This need goes far beyond the superficial graphical user interface design issues such as the arrangement of menus and the appearance of buttons. At issue are questions about how software professionals work, what tools they already know and use, how they understand the notation, and what (human) performance bottlenecks might profitably be addressed by language-based editing systems.

Asking user-centered rather than technology-centered questions casts new light on the design of these systems, their internal software architecture, services offered to users, configuration mechanisms, styles of interaction, and integration with other tools in the working environment. It also raises new questions about the underlying language-based technology, its usage, and what problems remain to be solved.

[1]SMARTSystem is a registered trademark of the ProCase Corp.

1.1 Thesis and Scope of This Research

The thesis of this investigation is that the success of language-based editing systems has been limited by inattention to user-centered issues concerning the context in which the tools are needed. When reconsidered from this perspective (drawing on evidence from software engineering practice, human factors, and empirical experience), many usability problems can be seen in past systems:

1. Annoying restrictions on text-based editing, the style of interaction most people prefer.
2. Failure to address the real productivity bottleneck, the difficulty people have understanding programs.
3. Exposure of underlying language-based technology to people, leading to inappropriate user models of system state and document structure.
4. Monolithic document presentations that fail to exploit all the information available in the programming environment.
5. Brittle system behavior in the presence of ill-formed programs or inconsistent system information.
6. Awkward (or missing) mechanisms for incorporating new languages.
7. Inflexibility because of closed data models and weak extension facilities.

Each of these problems has been partially addressed by earlier systems, but usually at the expense of other problems on the list. The tradeoffs are complex, however, and successful solutions must take the entire list into account.

This investigation begins with the following observation about these usability problems:

> The design of a language-based editing system presents a *user interface problem*, not only between user and system (in the conventional sense of the term) but between user and software documents (the more important meaning in this context).

With the design challenge posed in this way, new questions must be asked and new balances struck on fundamental choices such as text- versus structure-based interaction, language-specific versus generic services, and frequent versus infrequent analysis. The application of *user-centered system design* principles [92] casts new light on the issues and suggests that a well-founded, principled, and coherent approach to the design of language-based editing systems is possible.

This dissertation presents a new design framework for language-based editing systems, one that begins with a basic separation of design concerns: internal document representation and analyzer implementation; configurable, language-independent mechanisms to support user interaction with documents; a coherent and flexible set of user-visible system features and policies; and adaptation of the system to particular working contexts. *Pan* is a fully implemented, multi-lingual, language-based editing and browsing system developed at the University of California, Berkeley [12,31,132] that embodies this framework and demonstrates the viability of the approach.

An important goal of *Pan*'s design framework is to "decouple" user interaction in these systems from the linguistic and implementation details of their enabling technologies, and in particular from the compiler-oriented approach that has dominated their design in the past. Users do not think of programs in the same terms that are useful for designers of compilers. For example, a person confronted with a new program first reads comments and then examines the names of program entities [20], for example procedures and variables. A compiler confronted with the same program first discards comments and then abstracts away all names. This example is not meant to imply that a language-based editing system should analyze comments, but it does suggest the depth of the conceptual challenge for designers who seek appropriate application of the technology.

Adaptation to working contexts is captured in *Pan*'s design framework by the notion of multiple *view styles* for user interaction, each specialized for a particular combination of user population, task at hand, and language being used. A view style: (a) includes traditional syntax and static-semantic language descriptions, but may extend to extra-lingual analysis such as stylistic and usage guidelines; (b) specifies services to be provided and specializes generic services for the particular language; (c) defines a visual context, including typography and use of color; and (d) configures details of interaction, including keystroke and menu-bindings. A human designer creates view styles. A working *Pan* system includes a suite of view styles that collectively offers appropriate services and uniform user interaction. *Pan*'s design framework provides tools, guidelines, and examples, among which are solutions to usability problems that plague earlier generations of systems.

Finally, *Pan*'s design framework is *open*. To realize the full power of language-based interaction, the editor must function as an interface through which an open-ended collection of language-related services can be delivered to users. Known as *applications* in the *Pan* framework, these additional services can be added to *Pan* using its extension language, rich configuration mechanisms, and an extensible data repository. Alternately they can be delivered by integration with other tools, for example allowing *Pan* to serve as a user interface for compilers, profilers, and debuggers.

1.2 Overview of Dissertation

Chapter 2 begins with an introduction to the context of this research: the intended users, the class of tools involved, and the ultimate role to be played by such tools. The chapter then reviews the list of usability problems identified above, bringing outside perspectives to bear on the history of language based editing systems. The software engineering perspective casts light on the working environment and the nature of software systems. The individual perspective examines what is known about the mental processes of the people who carry out those tasks. From the list of shortcomings are derived a set of proposed requirements for language-based editing systems that will correct those problems, requirements addressed by the prototype system designed and built for this research.

Chapter 3 introduces *Pan*, beginning with a few short scenarios that show it in action. An introduction to *Pan*'s design framework follows: guiding principles, a four-layered architectural model, and five design metaphors that capture the spirit of the system from the perspective of its users.

In Chapter 4 the design and construction of *Pan* are described in more detail, addressing in turn each layer in the architectural model, beginning with the lowest: (1) language-based infrastructure; (2) kernel mechanisms; (3) elements of user interaction; and (4) view style design. This model enables an implementation that delivers language-based functionality without allowing the design of the analyzers to dictate models for user interaction. This chapter presents a "structural" description of the system, reflecting a point of view concerned with implementation: how the system is constructed and how it works. Sections 4.1 and 4.2 focus on internal issues, Sections 4.3 and 4.4 on those aspects presented to users.

Chapter 5, in contrast, represents a "functional" description of *Pan* that focuses on users and their tasks: what the system can do and how it can be used. Each section examines a particular aspect of user interaction: potential benefits with reference to the goals discussed in Chapter 3 and challenges to effective implementation. Some of the challenges result from conflicting goals, others from particular language-based technology. In each case *Pan*'s framework provides mechanisms, services, and design guidelines to solve the problems in accordance with general principles of user-centered design.

- **Text Presentation** An important step toward enhancing program comprehension by users is to exploit the full potential of human visual bandwidth; Section 5.1 describes how this can be done using a high-quality, configurable, text-based display. A unifying theme for *Pan*'s text-based services is to provide as many visual channels as possible for the conveyance of *meta-information*, that is, information *about* programs being viewed.

- **User Models** Section 5.2 turns to problems that arise when browsing and editing services operate over a domain whose structure is not isomorphic to the visual field presented to users. This is a particular problem for language-based editors, where the customary approach is to display programs in the conventional two-dimensional textual field, but to provide operations over the domain of a *tree* that represents an abstraction of a program's formally-specified context-free syntax. *Pan*'s design supports a model that more closely approximates how programmers actually think about program structure, thereby avoiding many usability problems associated with earlier systems.

- **Mixing Modes** A commitment to unrestricted text editing combined with language-based operations runs the risk of exacerbating the problem of hidden state: information in the dynamic editing context that is important to the user, but which has no direct visual analogue. These problems are minimized in *Pan*'s design by presenting to the user a model of language-based services that appears to be a simple extension of the familiar text-based model, accompanied by a metaphor to explain the relationship between the two. A *Pan* user may mix text-based and structure-based operations freely. Experience with this model, combined with further task-based analysis, has led to a novel design for *structural selection*; the new model promises to simplify user interaction and to permit convenient delivery of many useful services not otherwise possible.

- **Ill-Formedness** Unrestricted text editing, an important commitment in *Pan*'s design, implies that a program being modified is often ill-formed with respect to the definition of the formal language in which it is written. Many language-based editing systems treat these as aberrant situations, and they restrict or otherwise penalize the user until the problems are corrected. In *Pan*'s design ill-formed programs are presumed to represent the normal case, and services are designed to keep operating as usual. In fact, it is crucial that one distinguished set of services, those concerned with diagnosing and communicating about language errors, continue to be available whenever the user requests them, as described in Section 5.4.

- **Incompleteness** Section 5.5 shows how incompleteness, a special case of ill-formedness, can be addressed by *Pan*'s variant on strict structure-based or top-down editing: *placeholders* and *templates*. Like all *Pan*'s services, these add to the user's options rather than restricting them.

- **Inconsistency** A more subtle problem with the commitment to unrestricted text editing arises in the form of difficult choices about when and how to attempt incremental reanalysis following textual modifications. These choices involve issues as diverse as the cognitive structure of the user's tasks, the ability of analysis algorithms to recover from program errors, and the speed of computational platforms. The approach taken in *Pan*'s design is to adopt a lazy policy, deferring analysis until the user requests it, either explicitly or implicitly, by invoking an operation that requires current information. In the interim, between a textual modification and a subsequent reanalysis, the program state in *Pan* is said to be *inconsistent*. Section 5.6 describes several approaches that help make this state tractable for users.

- **Views** Section 5.7 shows how more than one *view* of documents can be useful to users. An alternate view in *Pan* typically presents some subset of the information available about a document, organized and presented in a form helpful for particular tasks. Connections among all views of a single document, however, permit the user to associate the different kinds of information when needed.

- **Applications** In Section 5.8 a number of more advanced services are described. These services have been or could be implemented using the basic elements of *Pan*'s design. The breadth of these applications demonstrates the flexibility of *Pan*'s model for language-based interaction.

A reflection on the success of the *Pan* prototype in meeting its research goals appears in Chapter 6. For example, some aspects of user interaction have shown themselves to be unsatisfactory in the current prototype, based on both experience with the system and ongoing analysis. An important kind of insight gained through work on *Pan*'s infrastructure concerns the degree to which language-based mechanisms must be recast in pursuit of user-centered goals.

Finally, Chapter 7 summarizes the lessons learned during this research, and includes suggestions concerning promising areas for further work.

Chapter 2

Background

This chapter introduces language-based editing, with particular attention to the context in which these systems are intended to be used. Evidence about that context, drawn from software engineering practice and from the study of computer human interaction, confirms empirical observations that usability problems abound. These problems are discussed here in general terms, and requirements are articulated for a system that corrects the shortcomings mentioned in the introduction.

2.1 Context

This section describes the context for this research: the intended user population and the class of tools being studied.

2.1.1 The User Audience

Software professionals manage large collections of interrelated documents. Many traditional assumptions about traditional language-based editors do not hold in this domain, for example that the main advantage of such systems is to help beginners avoid syntax errors, and that restricting users is a reasonable cost to pay for that advantage.

In terms of Neal's model of expertise among users of program editors [90], these professionals have high levels of expertise with computers, and usually have high levels of expertise with both programming and with languages. There are situations in which these users share some characteristics with beginners, for example when learning a new language, but it is important that design compromises not be made to accommodate low levels of general and programming expertise.

The working context is further enriched by familiarity with many existing tools, their functionality and their interface characteristics. New tools will be rejected if they merely duplicate existing functionality, if they operate needlessly unlike others, or if they cannot share data with other tools.

2.1.2 Language-Based Editing Systems

The term **language-based** indicates that one or more system facilities exploit language-specific information derived from documents. In the context of this research, the term

system (or **editing/browsing system**) encompasses all services used to browse, manipulate, and modify one or more documents interactively.

In the broad view there are many such systems, beginning with text editors that support word- and paragraph-based operations, including spelling checkers, and continuing through document preparation systems and outline processors. The success of spelling checkers (measured by their widespread use) offers a suggestive analogy for the design of language-based tools: they usually operate only when asked, and they carefully defer to the judgment of the user. The implicit assumptions are that a spelling checker is clerical but fast, must be given a dictionary, and cannot be assumed to understand anything about the *intent* of the author.

For software documents, linguistic knowledge available to the system can be shallow (for example language-modes in EMACS exploit shallow lexical structure) or deep (for example *cliché*-based editing in KBEmacs, which exploits deep plan-based structure [106]). At the limit of currently feasible technology are systems that have knowledge of the syntactic and static-semantic structure of software, and perform some browsing and manipulation using that information.

Proponents of language-based editing appeal to the compelling intuition that a system knowledgeable about underlying languages could help users with many of the demanding aspects of programming. Bernard Lang, coauthor of the influential Mentor system bases his testimonial on considerable experience with Mentor: "For advanced users, a syntax directed environment is irreplaceable for program maintenance, especially when used to the full capacity of this technology, with the ability to use all tools and program new ones in terms of the syntax directed paradigm, i.e. on the basis of the abstract syntax of the manipulated languages" [71].

At the time this research began, a number of language-based editing systems had already been implemented or designed. Following Ballance's taxonomy [9], those can can be roughly classified into three categories based on underlying models of editing: display-oriented text editors such as EMACS [122], syntax-directed structure editors such as Mentor [29], and syntax-recognizing editors such as Babel [57]. The rest of this section describes the three categories in more detail, pointing out the design tradeoffs they represent. Each category in the list reflects progressively more ambition with respect to services delivered, along with correspondingly greater design challenges.

How a system captures and maintains structural information involves an important distinction. It can be done *explicitly*, by interpreting user actions during document construction, as in an outline processor where the user creates new sections explicitly. Alternately it can be *derived* by examination of the document after construction by the user.[1] The choice affects both underlying implementation and the design of the user interface. For example, *syntax-directed editors* rely on the user to construct structures explicitly. *Syntax-recognizing editors* attempt to derive a structural representation for the document. A purely *text-oriented* editor makes no real use of linguistic structures at all; its "language-based" operations examine only the surface representation of a document, perhaps inferring a very simple syntactic decomposition.

[1] The two approaches can coexist; for example Mentor can derive structure initially from a textually represented document, but subsequent editing can only be structure-based [29].

Text-Based Editors

A **text-oriented editor** operates on a document modeled both as a stream of characters and as a two-dimensional plane of characters. (The coordinates in this plane are commonly called "lines" and "characters within a line".) Users are free to operate on any character at any time.

While groups of characters, for example words and lines, might be accorded special status and operations, no special structural constraints are imposed on the document and few well-formedness constraints are enforced. Bravo [70], EMACS [122], MacWrite [2], and Z [140], are examples of text-oriented editors. Both EMACS and Z provide some language-specific editing operations.

The technology for text-oriented editors is relatively mature. Language-based editing operations in these systems present no great technical obstacles, but the level of service is quite limited.

Structure and Syntax-Directed Editors

Structure editors present a document as having a definite internal structure, with editing operations modeled as operations upon that structure. Most often, the document is tree-structured, with operations defined on subtrees. Outline processors are structure editors for tree structures; spreadsheet editors are structure editors for tables. Structure editors may or may not interpret the structures that they edit. When they do attribute language-specific meanings to the structures, they are often called syntax-directed editors.

A **syntax-directed editor**[2] is a structure editor that requires that the document be syntactically correct at all times: editing operations must "follow" the syntax of the language. As in a pure structure editor, the user can add new material to a document only at those points where it can be successfully grafted onto the existing structure. ALOE [84] and the Gandalf editors [94], Centaur [17], the Cornell Program Synthesizer [125] and the Synthesizer Generator [104], Emily [52], Mentor [29], PSG [7], and the SbyS editor of the Mjølner project [87] are all syntax-directed editors.

Structure and syntax-directed editors are considerably more complex to build than text-oriented systems: language-based machinery must be added, and a complex relationship must be maintained between text-based visual presentations and internal representations more amenable to language-based analysis. On the other hand, design of these editors is simplified by insisting that users express changes in terms of internal representations rather than in terms of the textual presentations.[3]

Syntax-Recognizing Editors

A **syntax-recognizing editor** [21] derives structural information from text, in order to check for correctness without necessarily demanding the consistency constraints of a syntax-directed editor. Syntax-recognizing editors can provide structural operations as well as text-oriented editing. Babel [57], the Saga editor [67], SRE [21], Syned [40], and the UQ editors [138] are all syntax-recognizing editors, as is *Pan*, the platform for this research.

---

[2]The term "structure-oriented" [93] is sometimes used in place of "syntax-directed."

[3]Some *hybrid* syntax-directed editors, including the Gandalf editors, the Cornell Program Synthesizer, and PSG, permit text-editing in limited contexts.

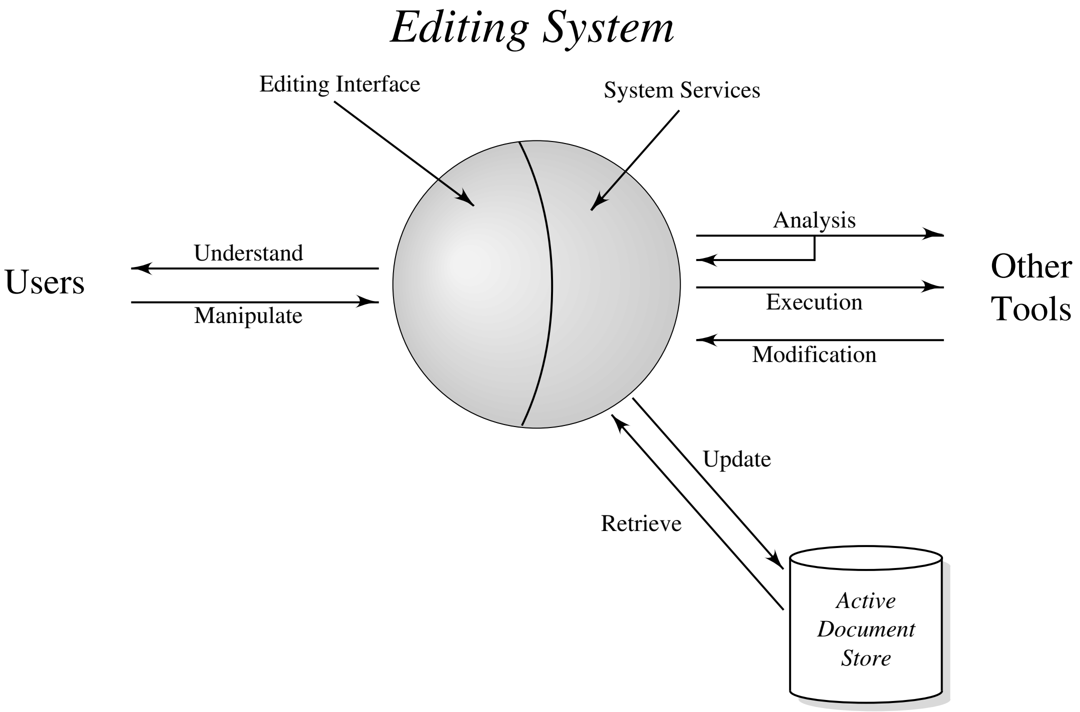

Figure 2.1: Editing interface and system services in relation to the environment

The syntax-recognizing approach[4] permits text-oriented editing by users at any time in any context, but at the cost of considerable complication in document representation, incremental analysis algorithms, and user-interface design. Common problems in early syntax-recognizing editors were a failure to hide this complexity from the user and limitations on language-based services when compared with syntax-directed editors.

2.1.3 The Role of The System

Early language-based editing systems were characteristically designed as standalone tools, although a few were integrated with a debugger and run-time system. In practical software engineering contexts a shift of perspective is necessary. The true utility of such tools is as the primary *interface* between people and integrated environments containing the documents they manage. Positioned this way, between user and documents, the system is uniquely situated to share information *about* documents that may be provided by both user and tools, as shown in Figure 2.1.

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. Useful tools in this model are characterized by information, possibly rapidly changing, *about* the code. Examples include debuggers, slicers, profilers, text coverage analyzers, and higher-level assistants like the cliché-based reasoning supported by KBEmacs [106].

[4] The syntax-recognizing approach does not preclude a user-interface that simulates syntax-directed editing. In fact, syntax-directed editing can be provided easily in a syntax-recognizing editor.

Complex, expensive analyses to support an editing interface make sense only in an environment in which many tools share the information maintained by the system. The same type checking that is used to tell the user that a document is type-correct can provide type information to a compiler, an interface consistency service, or an auditing tool. In some cases information produced by other tools should be made available to the editing interface. For example, the results of performance analysis or information derived from version control can be used to produce helpful views of programs or prototypes.

2.2 Usability Problems

This section returns to the list of shortcomings identified in the introduction, explaining the origin of each and associated difficulties for users. From the discussion emerge the design requirements pursued during this research.

2.2.1 Text-Oriented Interaction

Usability Problem 1 Annoying restrictions on text-based editing, the style of interaction most people prefer.

The history of language-based editors is marked by disagreement over text-oriented editing and the role it should play in language-based editing systems. Text remains the display medium of choice for documents in most languages, since most languages are by definition textual. Graphical presentations of such documents have some potential value for overviews and summaries, but it is very unusual for a graphical display to carry *all* the information present in a textual rendering, and it is seldom a natural visual field in which users can specify editing operations.[5]

The most fundamental requirement for language-based editing systems, discussed in this section, is thus the most controversial:

Design Requirement 1 Familiar, unrestricted text editing.

The Structural Hypothesis

Following the lead of the purely structural Emily system [52], one of the seminal language-based systems was designed around what will be called here "the structural hypothesis:"

> Programs are not text; they are hierarchical compositions of computational structures and should be edited, executed, and debugged in an environment that consistently acknowledges and reinforces this viewpoint. The Cornell Program Synthesizer demands a structural perspective at all states of program development [125].

Despite a few early arguments to the contrary [134,140] designers of language-based editors accepted the hypothesis and continued to impose varying degrees of restrictions on text-based interaction. Most designers argued that prohibiting "incorrect" operations would help users write programs faster or that it would enable the system to provide better services.

[5]This remark does not apply, of course, to those languages which are by definition graphical. For those languages, graphical display is the medium of choice.

One problem with the structural hypothesis is that language-based structure is not uniquely defined for a given language, but reflects instead a set of implementation choices based loosely on a formal language definition. For example, many early editors that provided unrestricted text editing maintained a complete parse tree [57,67], and there are many degrees of freedom available for the design of parsing grammars. Syntax-directed editors, on the other hand, usually maintain a reduced or **abstract** syntax tree (an example of each appears in Figure 2.2 on page 15). Arguments for the latter representation include the observation that it is a more "natural" model for user interaction, since the extra information in the parse tree is only an artifact of parsing technology. In practice, however, abstract tree representations are designed to meet the needs of tool implementation, not people.

A second problem with the structural hypothesis is that every operation burdens the user with the cognitive overhead of a complex relationship between the tree and its two-dimensional textual presentation. Lang mentioned evidence of this problem with Mentor:

> We must however note that, even with a good user interface, the syntax directed paradigm is too complex and bothersome for inputting programs or for performing simple editing tasks. As might be expected, it is particularly inconvenient for editing text/program fragments that are non-structured (strings, comments) or poorly structured (expressions) [71].

This kind of experience with fine-grained structural interaction led to the adoption of the hybrid approach by a number of systems, including the Cornell Program Synthesizer, in which small structural fragments may be edited textually.

Behind this second problem is serious clash between the editing model (what the user can do) and the presentation model (what the user sees). One can imagine a similar clash arising from a text editor that required sentences to be constructed only in terms of an underlying natural language grammar. Minör's attempt to correct this problem without abandoning the structural hypothesis led away from a text-oriented interface entirely for the SbyS program editor [87,88].

A third weakness is the implicit assumption that the only interesting structural categories are those corresponding to nonterminals in an underlying abstract grammar. But programmers' tasks involve many kinds of information beyond the purely syntactic, as will be discussed in more detail below.

Experience with Structural Editing

Experience confirms these arguments against the structural hypothesis. For example implementors of Rita, an editor for structured documents, were forced to augment their purely structure-oriented user interface with text-oriented operations because:

> Many editing operations which appeared simple and straightforward from the user's perspective, could involve fairly complex keystroke sequences because of the nature of tree-processing operations. Although most users understand that there are rules governing the structure of a tagged document, they tend to view a document as a linear progression of text rather than as a hierarchy of document elements [27].

Authors of two important language-based system for programs, PSG and Gandalf, have likewise concluded that text-oriented editing should not be limited [8,24].

Lisa Neal examined how people react to this kind of system, concluding that "very experienced users will reject any tool that does too much for them. For instance, they will only use a tool which allows free text input but provides error checking capabilities" [90].

What Users Want

In practice, people will not sacfifice text-oriented interaction. It fits naturally with the visible nature of the notation, and people are accustomed to it. Furthermore, most kinds of documents contain textual chunks that have no structural properties beyond the textual: examples include sentences or paragraphs in most natural language documents, labels in spreadsheets, and comments in programs. Finally, in a world of multiple languages text remains the least common denominator, a fallback position for users that is necessary when language-based systems produce different modes of interactions for each different language.

2.2.2 The Comprehension Bottleneck

Usability Problem 2 Failure to address the real productivity bottleneck, the difficulty people have understanding programs.

Early arguments for language-based editing systems focused on the need to expedite entry of syntactically correct programs. For example, one advantage cited for grammar-based template expansion is that it saves keystrokes. The COPE system uses *predictive data entry* during textual entry; it performs very elaborate error repair that allows the user to enter somewhat abbreviated code [4]. Unfortunately systems tuned for expeditious entry fail to address real obstacles to productivity.

Mary Shaw, reflecting on software engineering, observed that "Each order of magnitude increase in the scale of the problems being solved leads to a new set of critical problems that require essentially new solutions" [114]. An important example is that 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 [45, 139]. This observation leads to the next requirement for language-based editing systems:

Design Requirement 2 Rich information display.

Visual Presentation

One approach to enhancing comprehension draws on established traditions of graphical design in order to exploit the full power of the textual medium. The value of high-quality typography for natural language documents is well established, and recent studies suggest the same potential benefits for programs [6,95].

These studies, however, involved printed versions of programs typeset onto paper from source code using language-specific formatters, a context that differs in two ways from that of language-based editing systems. First, color produced by CRT displays evokes different perceptual effects than it does on paper [33], and other characteristics of the CRT medium have observably detrimental effects on reading speed [47].

Dynamic Display

A second difference between paper and a language-based system is the dynamic nature of the context: additional kinds of information (cursor location and the like) must be displayed, and users can potentially control what information gets displayed. The user's needs for dynamic display can be seen by considering in more detail what it means to read software.

Programmers read because they spend most of their time engaged in software maintenance, where the essential task is understanding existing software; this is called "design recovery" when some of that understanding is to be recorded [15].

Evidence suggests that reading software is a cognitively active process and has a fine-grained task structure [68]. The reader repeatedly forms hypotheses, which are then confirmed or denied by further reading. This happens opportunistically, using a variety of information when it is available. Reasoning tends to alternate between forward-chaining and backward-chaining, depending on the information available [74]. This suggests that a language-based editing system's display be visually rich and dynamic at the same time.

Even when writing new documents, however, programmers spend most of their time reading what they've just written. This is an essential characteristic of the notation and the process of writing in it. Writing software is a creative design process, and like many kinds of design it is done iteratively, with cycles of explicit interaction and feedback from what has been committed to notation so far. One finds similar recursive interaction between author and expressive medium in many creative endeavors [60], as well as in the work of Schön's "reflective" practitioners [110]. So one aspect of the author's conversation with the notation is a stream of questions like "Where am I now?" "What are the implications of what I've done so far?" and "What's left to do?" This argues all the more strongly for a rich, flexible information display (more about this below).

2.2.3 Technology Exposure

Usability Problem 3 Exposure of underlying language-based technology to people, leading to inappropriate user models of system state and document structure.

Some of the difficulty users encounter with language-based editing systems can be traced to inattention to principles of user-centered system design [92]. In particular such systems often fail to present a coherent conceptual model of system behavior that helps users apply the system to their tasks.

Given the role proposed earlier for language-based editing systems, as interface between user and software documents, it is equally valid and ultimately more important to apply the same principles in a second way. Coherent interaction between users and documents can be achieved by presenting conceptual models of document structure that help users understand them. This section looks at both kinds of interaction, with document structure and with system state, and sets forth the next requirement for an effective language-based editing system.

Design Requirement 3 Coherent user interaction with system and programs.

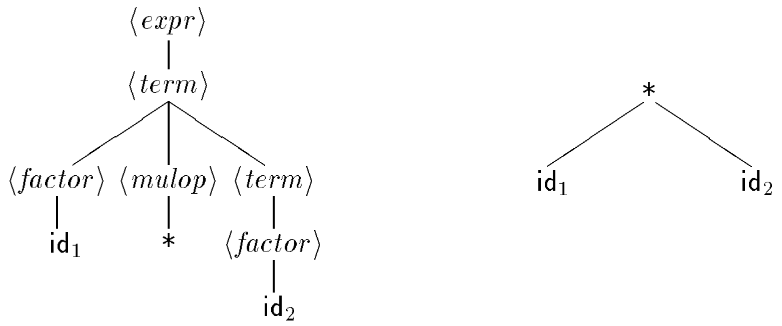

Figure 2.2: Parse versus Abstract Tree Representations

The Structure of Software Documents

Making language-based structure intelligible requires that the system present document structure in terms of coherent conceptual models. Most language-based editors present a document model based implicitly on the way they represent documents internally, typically as a tree. For example, many early editors that provided unrestricted text editing maintained a complete parse tree [57,67], whereas syntax-directed editors usually maintain a reduced or **abstract** syntax tree (an example of each appears in Figure 2.2). Arguments for the latter representation include the observation that it is a more "natural" model for user interaction, since the extra information in the parse tree is only an artifact of parsing technology.[6] Some of the problems with this approach were mentioned earlier in the critique of the structural hypothesis, namely that typical internal tree representations are not well-defined by languages, that they can cause confusion when used as the primary editing model, and that they don't correspond to the user's ideas of relevant structure. Consider for example the parent node $\langle expr \rangle$ of the parse tree in Figure 2.2 and its child $\langle term \rangle$. Both correspond to the expression that the user sees as "id_1 * id_2", the distinction is only an artifact of parser and perhaps analyzer technology, and simple structural navigation from one to the other would produce no apparent change in location.

From a user's perspective software documents are 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 that supported such a multiplicity of structural aspects could confuse users, but it need not if the system supports those structures already understood by users. People routinely think about complex objects from different perspectives and are remarkably adept at shifting perspective. An effective editing interface need only support these shifts without imposing any extra overhead on the user.

The formally defined syntax of a language, and closely related internal tree representations, represents a good starting point for exploring aspects of software structure. It provides a "backbone" decomposition of documents into structural components that help the system associate useful properties with regions of text on the screen. Even here, however, many aspects of an internal representation (many kinds of tree nodes) may *not* correspond to any useful conceptual structure.

[6] An abstract tree also requires much less storage than an equivalent parse tree, a difference that can translate into improved system performance.

More interesting structures tend to involve non-local and non-hierarchical relationships among document components. For example, consider the relationships in a natural language document defined by the connection between a figure and references to it, the relationships between declarations (definitions) and uses of variables in a computer program, the structures represented by a call graph in a program, or all uses of a particular code library. These are examples of relationships that can be derived by language-based analysis, and which can then be used for browsing, for querying, and by treating them as links for the purpose of non-local navigation in the style of hypertext.

Other decompositions can arise from non-derivable design information that the user adds. For instance, editors like ED3 [123] and the Cedar editor Tioga [127] allow users to specify explicitly a hierarchical decomposition for each document, where the structure may or may not be correlated with structures in the underlying formal language. Outline processors are editors for precisely this kind of structure where the underlying language is simple text.

Some kinds of decomposition fall into the area between linguistic and non-linguistic. Such structure typically concerns how a language is used, and derives from the design of particular systems and from coding conventions they use. These carry a great deal of the information that a programmer needs, and may be partially derivable, for example, using analyzers that can identify uses of locally proscribed language features, and can reason about stylized patterns of use. These kinds of analysis move far beyond simple error checking supported by earlier language-based systems: they involve knowledge of particular organizations, techniques, and systems.

User Models of System State

Just as coupling the user's model of document structure too closely to underlying implementations can make documents hard to understand, confusing behavior can result from excessive coupling between important aspects of a system's interactive state and its internal representations. Here user-centered design of a more conventional sort can help. For example, it is important to make visible those aspects of the system's state that have direct consequences for user interaction. An example from the Synthesizer Generator Reference Manual [102, page 91] shows what can happen otherwise:

1. The user sees "`((2 + 3) * 4)`" and highlights (places the structure cursor at) the inner expression "`(2 + 3)`";

2. Editing textually within the focus (structure cursor), the user deletes the parentheses, leaving "`2 + 3`" in the focus.

3. The user invokes the command `forward-with-optionals` to move the structure cursor away from the change site; and

4. The system re-inserts the parentheses before moving the cursor.

This behavior is confusing and probably not what the user intended, hence a warning and this example in the manual. Just before the user invoked the navigation command in step 3, only a subtle line of reasoning (which involves taking careful note of the structure cursor placement) would help the user predict from the display "`(2 + 3 * 4)`" what would happen next. Had the structure cursor instead been placed during step 1 over the outer expression "`((2 + 3) * 4)`", the system would have done what the user probably intended; even then, however, new parentheses would mysteriously appear around "`3 * 4`". The problem in this case is that fine-grained details of the (invisible) internal representation have been allowed to exert influence over the system's behavior.

A related problem arises in systems that support structural cut and paste operations. For example it might seem reasonable to copy the list of identifiers appearing in the formal parameter list of a procedure definition and then to paste it into a call to that procedure. Although the two lists of identifiers might be textually identical and closely related conceptually, there may be sound implementation reasons for different internal representations. Many language-based editors would cause the paste operation to fail, and a usability challenge for those systems is how to either (a) explain the failure to the user, or (b) perform grammatical transformations based on what the user probably meant to do so that the paste operation can succeed [46,73]. Here again, some of the problems can be patched, but the fundamental weakness remains: exposing in the user interface too much of the irrelevant (to users) underlying technology.

2.2.4 Views

Usability Problem 4 Monolithic document presentations that fail to exploit all the information available in the programming environment.

As argued above, the true structure of software from the perspective of experienced professionals is complex and multifaceted. Different users and 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* [75,120] and *clichés* [106] are deep) to be useful.

In some cases, many kinds of *meta-information* can be superimposed on the rich text display of the document, for example by choice of font, coloration, and background shading. In other cases, reorganization and filtering of the information is more appropriate. The table of contents and index are examples of this approach for natural language documents; they assist document comprehension without adding new information. Given the multiplicity of structural aspects present in software systems, and the variety of meta-information that can be produced, for example by data-flow analyzers and performance profilers, the range of potentially helpful views is large. This leads to the next requirement for language-based editing systems, one that helps assure that users will derive maximal benefit from the rich sources of information present in the environment.

Design Requirement 4 Multiple, alternate views.

2.2.5 Service Degradation

Usability Problem 5 Brittle system behavior in the presence of ill-formed programs or inconsistent system information.

A persistent and general problem with language-based tools is that they fail to degrade gracefully in the presence of malformed, incomplete, or inconsistent information. To a user, however, these are the natural states for documents being managed—designers of tools should make every effort to continue supporting the user when documents are in these states. This observation leads to one of the most technically challenging requirements for a language-based editing system.

Design Requirement 5 Uninterrupted service.

Ill-Formedness

Inherent in the syntax-recognizing approach is acknowledgement that documents being modified are more often than not **ill-formed**: at variance with an underlying language definition. Ill-formed documents are often said to contain "errors," a pejorative term reflecting the limitations of many analysis methods. Many language-based editors that permit text-oriented editing inherit this bias. Unable to analyze ill-formed documents, these systems insist that the user correct any newly introduced "errors" before proceeding. Often justified as a service, because it limits the extent and duration of "errors," this treatment has unpleasant side effects.

- It narrows options available to the user, who may prefer to delay trivial repairs while dealing with more important issues. An "error" may often be part of an elaborate textual transformation.
- It implies that derived information is only available and accurate when documents are well-formed, again constraining the user.
- It implies that the user has done something wrong, when in fact the system is simply unable to understand what the user is doing [76].

Incompleteness

Incomplete documents represent a special case of ill-formed ones, but they correspond to natural intermediate states for documents being constructed by users and can potentially be treated more appropriately by language-based editing systems.

This challenge has been well addressed by technology developed early in the history of language-based editing systems. The first systems were so restrictive that users could only construct documents by invoking grammar rules starting from the goal nonterminal in the language, implicitly building a derivation of the desired program. The standard technique became known as the "placeholder," a visible glyph that appeared in a place corresponding to an unexpanded nonterminal in the derivation tree, usually the name of the nonterminal displayed in some way to distinguish it from terminal symbols.

The standard use of placeholders in syntax-directed editing is a very restrictive paradigm. Even in hybrid systems, placeholders are not editable by the same rules as the text.

Inconsistency

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. This results from design decisions to allow full text editing and not to perform analysis with every keystroke.

Inconsistency between text and derived information should not be confused with the related but separate issue of ill-formedness. Many language-based editors that permit text-oriented editing are able to resolve inconsistency only for well-formed documents. A user of such a system who has edited a document textually is not permitted to edit elsewhere in the document until those changes meet the system's requirements for syntactic well-formedness.

Inconsistency confronts the designer of language-based editing systems with a dilemma. Until the next analysis nearly all of the derived information (including diagnostics concerning ill-formedness) produced by analyzing the text is untrustworthy. On the other hand, not to support any language-based services in this state is a needless interruption of service, since a great deal of that information will continue to be correct most of the time.

2.2.6 Multi-language Support

Usability Problem 6 Awkward (or missing) mechanisms for incorporating new languages.

Software developers typically use several formal languages daily: design languages, specification languages, structured-documentation languages, programming languages, and small languages for scripts, schemas, mail messages, and the like. The construction of "little languages" is a respected programming technique [14], and these sometimes appear embedded in other languages. For example, many subroutine packages such as libraries for developing window-based applications effectively define mini-languages that determine how subroutine calls, and in particular long sequences of arguments, must be used.

Many language-based systems are built for a single language and cannot be easily adapted. Many that can be adapted still produce systems that can only handle one language at a time. Users cannot afford the time it takes to switch tools and change interfaces along with every shift in language: a language-based system must support all these different languages as smoothly and uniformly as possible. This accounts in part for the success of GNU EMACS, which has only the shallowest of language-based information available, but which can be extended and adapted easily for new languages.

Bernard Lang commented that:

> Language independence is essential for the adaptability of the environment to different dialects or to the evolution of a language. It is also a factor of uniformity between environments for different languages. ... However, even with a good language specification language, we believe that the definition of abstract syntaxes and finely tuned pretty-printers are non-trivial tasks that require much care and experience, especially if the language has been originally defined in terms of a concrete syntax [71].

It must be as convenient as possible to add support for new languages using natural, declarative, language-description mechanisms, which allow a description writer to focus on what is being described rather than on how document analyzers operate.

Design Requirement 6 Description-driven support for multiple languages.

Since the services that the system can provide are based on the data it derives from a document, a language *description* may include elements beyond the scope of the basic *definition* of a language. Thus, someone adding a new service to the system may need to extend some existing language descriptions to derive new data or maintain new annotations.

Finally, this issue presents a challenging design tradeoff between customization and adaptation for specific languages on one hand, and on uniformity of user interaction across languages on the other.

2.2.7 Inflexibility

Usability Problem 7 Inflexibility because of closed data models and weak extension facilities.

A usable language-based editing system must be flexible in many ways. As with most interactive tools, it must be customizable and extensible to accommodate the enormous variations among individual users, among projects (group behavior), and among sites. [71, 122].

An effective system must also be built on a flexible framework designed to accommodate many kinds of variation and evolution [51,71]. Programmability and an open architecture are essential here, permitting extension beyond that permitted by simple parameterization of existing services. The final, requirement for language-based editing systems suggests that as many degrees of freedom as possible be supported.

Design Requirement 7 Extensibility and customizability.

As an interface to tools in the environment, a language-based editing system must be capable of using a variety of information, derived by many different tools. Users opportunistically exploit many forms of information to help them understand and modify complex documents [74]. "Information gathering" is the primary task associated with important activities like program maintenance [56].

For example, many language-based systems check that a document is well-formed. The same analysis can enable a user both to edit the document in terms of its underlying language and to locate document components that violate restrictions in the formal language definition. Other kinds of interaction may require more elaborate analysis. For example, language-based formatting (sometimes called prettyprinting), traditionally based only on surface syntax, should exploit information about scopes, types, local usage, or even distinctions such as "main-line" versus "error-handling" code. Some kinds of interaction should be driven by profile information, by information from test generators, and any other tools in the environment that can produce potentially useful metainformation.

Chapter 3

The Pan System

The requirements derived in Chapter 2 are addressed by the design of *Pan*, a multi-window, mouse-based editing and browsing system that was developed to support ongoing research into integrated document development environments [12,132]. *Pan I* Version 4.0[1] [31], the currently implemented version of *Pan*, runs on SPARCstations[2] under SunOS and on DECstations[3] under ULTRIX, using the X window system [109]. Throughout this dissertation the term "*Pan*" refers to *Pan I*.

Pan I is the product of collaborative research. Two analyzers, *Ladle* [22] and *Colander* [9], provide the language-based information that drives *Pan*'s user-oriented services. Section 4.1 describes these analyzers briefly and discusses how they are integrated with the rest of *Pan*. A number of projects, some ongoing, have successfully used *Pan* as a development platform for research topics such as advanced textual presentations of programs [16], graphical presentation of data structures [98], and dynamic compilation by attributed transformational code generation [18].

This chapter introduces the *Pan* system, beginning with design requirements described in Chapter 2. A preliminary glimpse of the system itself follows, showing how *Pan* appears to its users when in operation. The remainder of the chapter discusses the approaches developed to guide its design and implementation.

3.1 System Requirements

Pan's design was motivated by observations about the nature of programming and the people who do it, and by drawing on the experience of previous generations of experimental systems with related goals. The design requirements listed in Figure 3.1 summarize the issues discussed in Chapter 2.

These requirements affect *Pan*'s implementation at every level, including the language-based analyzers that are otherwise hidden from view. One result of this research consists of insight into problems associated with the transfer of language-based technology from its batch-oriented and compiler-inspired origins into interactive editing and browsing systems. User-centered requirements such as these impose demands on the technology that diverge

[1]The version 4.0 architecture described here differs substantially from earlier *Pan* versions [9,13].
[2]SPARCstation and SunOS are registered trademarks of Sun Microsystems, Inc.
[3]DECstation and ULTRIX are registered trademarks of Digital Equipment Corp.

1. **Familiar, unrestricted text editing.**
2. **Rich information display.**
3. **Coherent user interaction with system and programs.**
4. **Multiple, alternate views.**
5. **Uninterrupted service.**
6. **Description-driven support for multiple languages.**
7. **Extensibility and customizability.**

Figure 3.1: Summary of Design Requirements for *Pan*

markedly from those imposed by traditional compilers. Section 6.1 returns to these issues.

3.2 Using *Pan*

Figure 3.2 shows a sample text editing session. On the surface *Pan* appears to users as a convenient bit-mapped, mouse-based, multiple window text editor. In the figure, two windows are open onto a text document being viewed, and a help window displays the keybinding configuration in that view.

Text editing services based on familiar models encourage smooth integration into existing working environments. Users familiar with EMACS [122] find the transition between the two editors smoothed by comparable text services and compatible keybindings [31]. Generalized unlimited undo, kill-rings, text filling, character classes, customization, extension, and self-documentation are among *Pan*'s many standard services. The same text-oriented services are provided for every document, whether or not its view style includes language-based configuration. 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.[4] *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.

Textual and language-based operations may be mixed freely, as shown in Figures 3.3 and 3.4. In Figure 3.3 the user has just typed in a line of new text (beginning with "`Fact :=`") by setting the text cursor with the mouse and typing. Note that the new line of text looks a bit different from the rest of the program, revealing that this text has not been analyzed and that *Pan* therefore does not yet have any language-based information about it. The user then invokes a series of language-based navigation commands that move the cursor forward a statement at a time[5] until the desired one is reached. As Figure 3.4

[4] In terms of Ballance's taxonomy [9] discussed in Section 2.1.2, *Pan* is a **syntax-recognizing** editing system.

[5] The command to move forward structurally is generic. The specific kind of structure for the command is selected from a menu, or by keystroke sequence, and appears in the panel as the "`Level`". The level

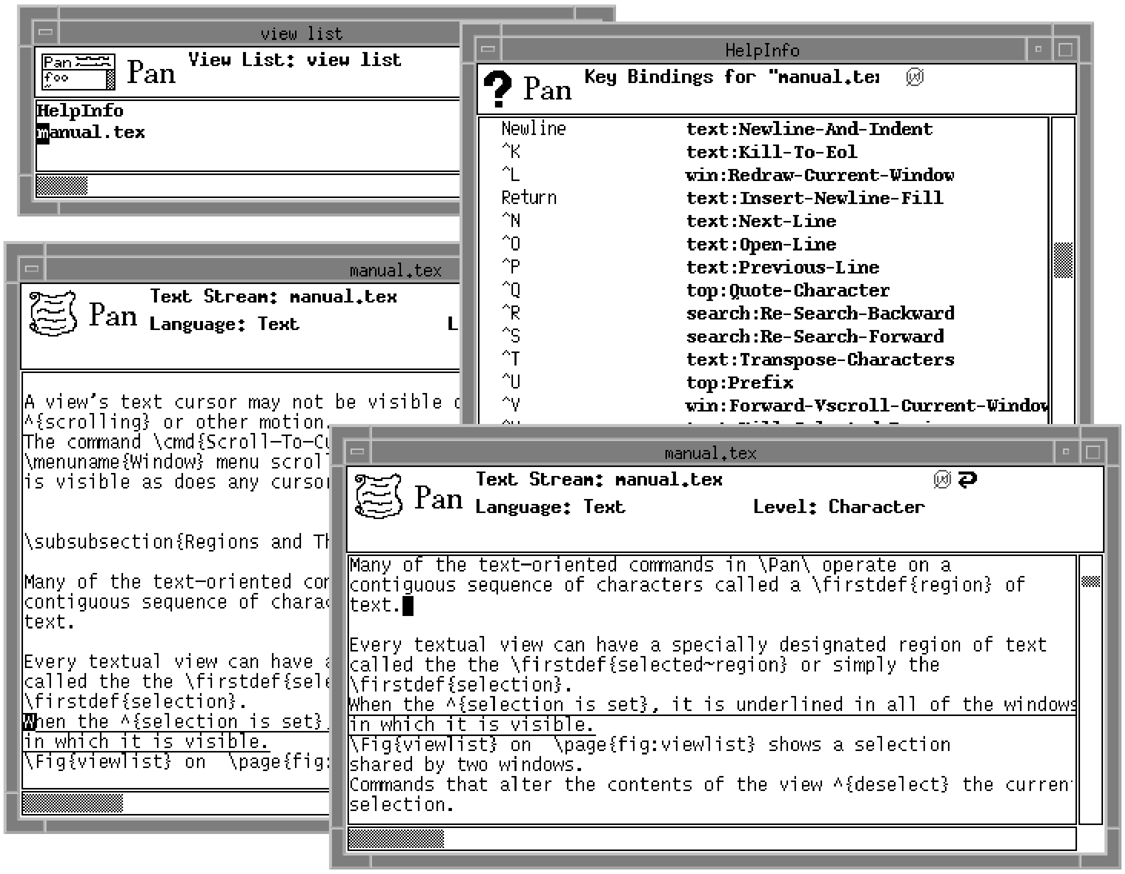

Figure 3.2: A *Pan* Session