

1

Productivity Engineering in the UNIX[†] Environment

AD-A173 780

Design of the Pan Language-Based Editor

Technical Report

S. L. Graham
Principal Investigator

(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Contract No. N00039-84-C-0089

August 7, 1984 - August 6, 1987

Arpa Order No. 4871

DTIC
ELECTE
S OCT 28 1986 D
A

DTIC FILE COPY

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

This document has been approved
for public release and sale; its
distribution is unlimited.

DISCLAIMER NOTICE

**THIS DOCUMENT IS 'BEST QUALITY
PRACTICABLE. THE COPY FURNISHED
TO DTIC CONTAINED A SIGNIFICANT
NUMBER OF PAGES WHICH DO NOT
REPRODUCE LEGIBLY.**

Accession For	
GRA&I	☒
TAB	☐
Announced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



Design of the Pan Language-Based Editor

Robert A. Ballance*

February 10, 1986

1 The PAN Editing System

PAN¹ is a language-based editing environment that will serve as the front-end to an integrated programming environment.

- PAN is multilingual. The system will support several languages within a single editing session.
- PAN is both text- and structure-oriented. Users are free to treat a document as either a stream of characters or a structured object.
- PAN performs syntactic and static-semantic checking.
- PAN is able to collect and maintain a database of information about documents. This database, updated incrementally throughout an editing session, is used by the semantic checker and by other tools that require information about a document or program.

Many systems have similar capabilities. ALOE[15], Gandalf[16], and the Cornell Synthesizer Generator[17] are multilingual. Babel[9], Saga[4], and Syned[7] handle both text and structure. The Synthesizer Generator[17] and Gandalf[16] are systems that provide semantic checking. How, then, does PAN differ?

*Copyright © Robert A. Ballance 1986. All rights reserved. Sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Naval Electronics System Command under Contract No. N00039-84-C-0089.

Robert A. Ballance was supported in part by a MICRO fellowship.

¹Why "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 by this project. 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 seems apt.

The primary design consideration for PAN is to support large system development by experienced programmers. This consideration leads to the following design goals:

- To provide both text- and structure-oriented manipulations while hiding the internal representation of the underlying structure from the user
- To retain information for other tools
- To provide as much feedback to the user as possible as soon as possible
- To support the experienced user
- To handle production languages such as C, Ada², and LISP
- To provide a test-bed for user-interface designs

The PAN system is viewed as a front-end for an integrated development environment. Algorithms and implementation methods sufficient for stand-alone editing systems are not always appropriate in the larger environment, where other tools need to share the information gathered by the editor.

In the final analysis, it is how any system supports these goals that distinguishes it from other systems. PAN has adopted the following approaches.

- Present the logical structure of a document.

Users of a language-based editor should be able to interact with the system in the terms and concepts of the language being edited. Thus, for a programming language having modules, functions, statements, and expressions, the user should be able to select and operate upon modules, functions, statements, and expressions. Text editors only provide operations on a stream of characters. Structure-oriented editors maintain an internal representation of the document's structure within which internal nodes represent the language's substructures. Those nodes, however, remain uninterpreted, requiring the user to use structure-oriented commands (e.g., Left, Right, In, Out, Delete-Node) rather than language-oriented commands (e.g., Next-Function, Previous-Declaration, Delete-Statement) to operate on the structures in the language.

PAN provides language-oriented, rather than structure-oriented, commands. The internal structure will be hidden from the user by the user interface.

²Ada is a trademark of U.S. Government—Ada Joint Program Office.

- **Integrated text- and structure-oriented editing**

Being able to interact with a document as either text or as a structured object is important for three reasons. First, documents, especially programs, have both text-like and structural aspects. For instance, a user will frequently search a program for uses of a particular identifier. Such searching may be a text-like operation, such as changing the name of an identifier wherever it appears. However, a user may also wish to make the change based on the structure of a program, for instance renaming a variable within a certain scope. In this case, variables having the same name in scopes nested within the given scope would retain their names. Second, documents contain unstructured text, be it the text of a sentence in a word processor or the text of a comment in a program. Third, there are times when a user wants to substantially transform the structure of a document and the most efficient way to do this is by altering the textual representation of the document. For instance, a user may wish to substantially rewrite and reorganize a document, reusing pieces already written.

- **Incremental LR parsing to detect syntactic structures in the document.**

New parsing and tree-building algorithms have been developed for PAN. These algorithms allow a compressed form of the parse tree to be used as an internal structured form for a tree-structured document such as a program. The internal form is related to the tree structure imposed by the language's grammar, yet may be substantially smaller than the parse tree created by a parser.

- **Static semantic checking based on logic programming.**

The model of semantic checking proposed for PAN is related to logic programming. (The best known example of logic programming is the language Prolog.) A database of information about a document is constructed incrementally during the editing session. Semantic constraints, expressed using clausal logic, are satisfied against the database. These constraints may be attached to structures in the language, or may be independent axioms of the language. Creation and use of the database are intertwined; certain conditions must be true before other facts can legitimately be added. The research involved includes the application of logic programming to static semantics and the search for efficient approaches to constraint satisfaction in the presence of a changing database.

- **Both keyboard (EMACS-like) and menu-driven operation.**

Using menus allows language-oriented commands to be parameterized according to the language being edited. There no need for a set of language-specific commands for each language. However, keyboard-driven operation is essential during certain phases of browsing and document-entry.

The remainder of this document is an overview of the design of the PAN system. The system has three major components: an editor (PAN), a language for describing languages to the editor (LADLE), and a processor for the language-description language. The system is being implemented using a mixture of LISP and C code. It runs on a Sun⁸ Workstation. Portions of the system are already implemented; the remainder are in various stages of design. This document was edited (in part) by the text-processing component of PAN.

2 System Components

The three components of the PAN system are a language-independent editor, a language-description language, and a language-description processor. The *language-independent editor* (called PAN) is the component that manipulates structured documents written in some formal language. Documents are often programs or fragments of programs, but other structured document languages (such as text-formatting languages) are supported. The editor can be customized for a specific language by providing a set of tables and functions that describe the language. The *language-description language* (called LADLE) is the notation in which descriptions are written; it provides a formal description of those aspects of a language affecting language-based editing. The *language-description processor* (called ladle) is a suite of programs that take a language description, check it for suitability, and build tables to be used by the editor.

Descriptions of computer languages that are manipulated by computers (including almost all programming languages) have several parts. The *lexical* specification describes the basic textual units (tokens) of the language. This specification can be regarded as a mapping from a stream of text to a stream of tokens. The *syntactic* specification describes the phrase structure of the language; this is a mapping from a stream of tokens to the substructures of the language. The *static-semantic* specification describes the context-sensitive aspects of the language, providing constraints that can be checked without actually executing the program. The *extra-lingual components* deal with issues such as handling comments, sharing of internal representations, emulation of preprocessors, file inclusion, and formatting. The *dynamic-semantic* specification describes the operational meaning of each executable construct in the language. The *operational* aspects of a language reflect implementation-dependent choices such as storage allocation and runtime memory structures.

This research is confined to the lexical, syntactic, static-semantic and extra-lingual aspects of languages. Other projects may create tools which use the information by PAN to implement other aspects of a language, including code generators, interpreters, and debuggers.

⁸Sun is a trademark of Sun Microsystems.

The PAN editor and the LADLE language-description language reflect the three-tiered structure (lexical, syntactic, static-semantic) described above. Extra-lingual specifications are handled in a variety of ways, each tailored to the specific problem being handled. Since the needs of the editor determine the content of language descriptions and the functionality of the description processor, the editor is the first component to be discussed.

3 Editor

PAN is an extensible, customizable editing system. The design of the text-oriented component of PAN has been strongly influenced by EMACS[18,19] and Bravo[13]. The influence of EMACS clearly shows in the presence of extension and customization facilities, and in the use of an active keyboard. Bravo's influence is seen in the use of mouse-and-menu operation and the concept of a currently selected region.

The facilities provided by PAN can be extended either by adding new language descriptions or by adding new commands. Extensive customization of the user-interface is also possible.

The subcomponents of the editor include

- Fundamentals: edit buffers, windows and keyboard handling, command dispatch, undo actions, help facilities, and extension and customization facilities.
- Text Representation : text processing commands.
- Token Representation: scanning, searching and editing on tokens.
- Tree Representation : incremental parsing, tree construction, and tree operations.
- Semantic Checker: database, assertions and constraints, and maintenance of consistency

3.1 Fundamentals

3.1.1 Edit Buffers

Edit buffers collect together the various attributes of an editing session. A single buffer is the locus of attention at any given moment. Almost all of the editing commands (selecting, viewing, altering, and traveling) operate on the currently active buffer.

A buffer is also the unit of interaction with both files and languages. There is one file associated with each buffer, and there is at most one language associated with each buffer. The text abstraction is a fundamental abstraction of every buffer.

3.1.2 Windows and Input

Windows in PAN have three levels of abstraction: a *window*, a *screen*, and a *physical-screen*. A window connects an internal representation to a screen and provides operations such as status and annunciator lines, scrolling, and cursor optimization. A screen contains information about the object currently being displayed. A physical-screen is an abstraction corresponding to a collection of pixels on a screen. Every screen has a single physical screen. Physical screens have been introduced to hide the particulars of any specific windowing system from the remainder of PAN.

The window subsystem of PAN is a mixture of LISP and C functions. LISP functions supply the higher-level window operations, such as scrolling, screen update, and cursor motion. The LISP routines may share the internal representations of a document with their related editors. C functions provide access to the *pixwin* level of the SunWindows⁴ window system. These are the functions that access and alter the display bit map. Screens are LISP-level data structures, while physical-screens are C data structures.

Input to PAN is also structured into levels. At the user-level, *Get-Character* returns the next keyboard or mouse event typed. Below the user level is a level that actually polls for the next keyboard or mouse event. This layer is the most interesting, since the polling mechanism can automatically invoke certain actions based on the amount of time elapsed since the poll began. In essence, the routine at this level executes a list of commands of the form

"If no input is received in the next X seconds, then execute Y"

When a keystroke is available, polling terminates and the key code is returned. If the command-list is exhausted, the routine enters the interactive wait state and the editor sleeps. At the bottom level is some C code for handling input and timer actions.

An example may sort this out: the outer command loop of PAN repeatedly asks for a keystroke at the top of the loop. The command to update the display is, in fact, executed from the work-list in the input routine. Thus if characters are available, full redisplay is inhibited. When keystrokes come more slowly, the screen is updated to reflect the most recent changes. This mechanism also allows execution of some commands based on "think-time."

3.1.3 Command Dispatch

Commands to the editor can be invoked in either of two ways: from a binding or from direct invocation. Execution from a binding is the most common. It occurs whenever the

⁴SunWindows is a trademark of Sun Microsystems.

user selects an item from a menu or types a sequence of keystrokes that has an associated function. Direct execution occurs whenever a function is invoked from another function and whenever a function is called by the function `Execute-Lisp-Line`.

Functions that are executed from bindings are not passed any arguments. However, all functions have access to the current environment, including the most recent key struck, the current cursor position, the current text or tree-node selection, and any numeric prefix arguments. Functions can also prompt for arguments.

The actual invocation procedure is complicated by the presence of *daemons*. A daemon is a function associated with (i) a command and (ii) a set of circumstances. The daemon is invoked whenever the associated command is executed and when the circumstances occur. If a daemon doesn't exist, no daemon-related action is taken. Daemons are not invoked when a function is called by another function.

PAN currently supports three kinds of daemons: *before*, *after*, and *undo*. *Before* daemons are executed immediately prior to invoking a function. *After* daemons are executed immediately after a function if the function returns normally. The results returned by the function form the arguments to the *after* daemon. *Undo* daemons implement undo actions.

The standard processing sequence can be altered in two ways: by suppressing invocations and by using the normal error sequence. Calls to `Editor-Error` short-circuit further processing and return control to the outer loop. In particular, no undo information is logged if an error is detected. Suppression of execution is attained by calls to `Suppress-Normal-Function` and `Suppress-After-Daemon`. These two calls affect only the current command being executed.

3.1.4 Keymaps and Menus

A keymap is a mapping from keystroke sequences to functions. Each function implements an editor command. Keystroke sequences can be one or two characters long. In the two-character case, the first character (called a prefix key) selects an alternative set of key assignments while the second character selects a key on the keyboard. There are currently four alternative sets of key assignments, including the normal (unprefixed) keyboard, and assignments prefixed by `Esc-`, `Ctrl-X`, and `Ctrl-Z`. Mouse buttons behave like ordinary keys.

Each buffer has its own local keymap. A global keymap defines key bindings visible for all buffers. Local bindings override global bindings.

Menus provide an alternative binding mechanism. A menu provides a series of slots that can be bound to functions. When a menu and slot are selected, the function bound to that combination is invoked. There are both local and global menus. Menus are named with symbols drawn from a global name-space.

3.1.5 Undo Actions

Undo actions are based on reversible editor actions. Suppose *Cmd* is an editor command. The undo daemon for *Cmd* supplies the undo action for *Cmd*. The arguments passed to the undo-daemon for *Cmd* are the values returned by *Cmd*.

The PAN editor supplies hooks for implementing any of several undo strategies[20,2]. A particular strategy can be introduced by rewriting the default Undo function. The default strategy is called "undo/undo"; one can only undo the most recent undoable action, a second call to Undo will reverse the undo action. This is the undo facility provided in vi.

After a command is executed from a binding, the results of the command (or the result of the after daemon if it was invoked), together with the command's name, are passed to the undo processor by Mark-Undo. When Undo is invoked, the appropriate undo daemon is called. The undo daemon receives the results of the command being undone as arguments.

3.1.6 Help Facilities

Help facilities are an integral part of PAN. The extension facilities compute and retain information used to provide online assistance.

The Help command is the primary entry to the system. It formats and prints information about systems facilities or particular functions.

The Apropos command provides keyword search through the commands defined in the system. Apropos uses a standard command naming convention (words separated by hyphens) to decompose command names into their component keywords. For instance, Next-Character is broken into "next" and "character"; (Apropos 'next) will then recall Next-Character, among others.

3.1.7 Extension and Customization Facilities

At heart, PAN is an open system allowing a knowledgeable user access to almost all of its functions. Users can extend PAN by adding new languages or defining new commands. A user is free to customize PAN by altering keystroke and menu bindings, adding new menus, or changing display attributes. By making PAN extensible, the initial implementation can focus on core concepts while assuring that the functionality required by a wide spectrum of users can be provided. By making PAN customizable, individual users can tailor the system to their own tastes, and designers of user interfaces can experiment with alternate interfaces.

PAN can be extended on two levels: by adding new languages and by defining new functions. To add a new language, one writes a new description in LADLE and processes it.

Once the tables created by the LADLE processor are available, it is a simple command to load the new tables and edit a file.

The facilities for defining new commands include

1. Command, flag, option, and variable definition,
2. Key and menu binding, and
3. Automatic loading of files and buffer-specific configuration (Auto-Load and Auto-Exec).

The command **Define-Command** is used to define a new command to the system. **Define-Command** accepts a variety of keyword arguments that specify various aspects of a command. The argument list may contain optional arguments. Values for optional arguments are supplied by the caller, by default values or by prompting the user for an input. The most common use of **Define-Command** will look like:

```
(Define-Command My-New-Command (arguments)
:help "The documentation retrieved by the HELP command"
:undo Undo-My-New-Command
...)
```

"My-New-Command" is the name of the command being defined. Here is a complete list of the keywords accepted by **Define-Command**:

```
:after The name of the after-daemon for this command.
:apropos Additional words to include apropos this command.
:before The name of the before-daemon for this command.
:help Short documentation. This field can be either a LISP string or a list of strings. Lists
of strings are printed one-per-line by Help.
:undo The name of the undo-daemon for this command.
```

PAN can be customized by reassigning keys, loading new libraries of commands, defining new options, and by setting the available options.

3.2 Text Representation

The text editor supplies two abstractions of text: as a *stream* of characters punctuated by newlines characters, and as a *contiguous region* of characters. The stream abstraction supports searching, cursor motion, and scanning (for tokens). Regions are the fundamental unit underlying operations that alter the contents of a text stream.

The primary data structure underlying both abstractions is an implementation of "sticky-pointers"—a data structure described by Fischer and Ladner in [6]. A sticky-pointer is a pointer to an item in a stream that remains stuck to that item as the stream is altered. Sticky-pointers are not updated as the stream is altered; instead they are updated when the pointer is dereferenced. This is done using a level of indirection that adds a small storage overhead, but defers updating pointers. Sticky pointers are used to implement cursors, marks, regions, and the mappings from tree to token and token to text. The implementation of sticky-pointers in PAN includes some storage optimizations not discussed in [6].

3.3 Token Representation

Tokens are an intermediate stage between a stream of text and a tree-structured document. Tokens are detected by a function called a *lexical analyzer* or *scanner*. Some tokens have a single textual representation (e.g., keywords), while others have many textual representations (e.g., identifiers or numeric constants). In addition, some tokens do not affect the structure of a document (e.g., comments). These tokens must be recognized by a lexical analyzer, but are not passed on to the parser.

Every token in the system has some fixed and some variable data associated with it. Fixed data include the integer identifier used by the scanner and the parser whether it is a single-name or multi-name token, and whether it is screened from the parser. Variable data include the token's position in the text stream, the token's parent in the internal tree, and possibly even its textual representation.

Tokens can be recognized using standard interface to lexical analyzers. PAN provides a basic interface that supplies functions for input, lookahead, and for calling a lexical analyzer. These functions follow the general interface used by lex[14]. In particular, the functions *input()* and *unput()* are provided. The lexical analyzer is expected to be an integer-valued function that sets the length of the token found in the external variable *yyleng* and that returns the integer id of the token as its value. Either lex-generated or hand-written analyzers can be used, provided that none of the symbols used generate name conflicts in the load module.

The tokens can appear either as leaves of the internal tree, or as attachments to nodes in the internal tree. Stream protocols (e.g., Next-Token) and editing operations at the token level are implemented using either text-oriented or tree-oriented commands.

One alternative to this approach is to represent the stream of tokens explicitly as a separate data structure. While retaining an explicit token stream is advantageous in some situations, doing so adds extra storage overhead.

The mapping from tokens to text is direct—each token will have a sticky-pointer to its image in the text stream. The mapping from text to tokens is approximate—each chunk of characters (a chunk is a line or less) will have a pointer to the token that includes the first character of the chunk. If the chunk begins with a character that is not part of any token, the pointer will point to the token whose last character precedes the chunk in the text stream. These pointers can be set when the document is scanned or when a text representation is generated from the tree.

3.4 Tree Representation

The tree representation is the structure-oriented portion of PAN that provides facilities for constructing and manipulating trees. Trees are the explicit representation of the structure inherent in many documents. In PAN, trees can be constructed either top-down, via a sequence of user-specified actions, or bottom-up by parsing a textual representation.

One design goal is to fully integrate text- and structure-oriented editing of documents written in a language that having a formal definition. To achieve this goal, the system must be able to recognize the tree-structure in a textual representation of a document. It must also be able to do the reverse: to generate a textual form from a tree. Generating a tree from text is called *parsing*. In an editing environment, it is undesirable to reparse an entire document when an alteration occurs. Parsing algorithms that limit their operations to affected areas of a document are *incremental parsers*.

A second design goal is to hide the physical structure of the internal tree. In most structure-oriented editors, the user is required to understand both the tree structure and its relationship to the structure of the underlying language. Operations in these editors are tree-oriented rather than language-oriented. Movement is restricted to follow the tree, node to node. The command set is node-oriented (Up, In, Right, Delete-Node, ...). However, in PAN, movement is language-oriented (Next-Statement, Delete-Declaration ...).

Adopting the fully-integrated, text-and-structure model has two drawbacks, however. First, constructing a structure tree directly from the text often results in a large and complex tree. Second, there is no guarantee that the text of a document will represent a well-formed tree; the document may well be syntactically incorrect. In the next sections, we discuss our solutions to those problems.

3.4.1 Parsing and Error Handling

When a language is manipulated by a computer program, two views of its syntax become relevant. The first, called the *concrete syntax*, is the syntax used when analyzing sentences in the language. The concrete syntax is described by a context-free grammar which is used by a parser-generator to create a parser for the language. The second, called the *abstract syntax*, describes only those structures in the language having independent semantic relevance. The concrete syntax is larger and more complex than the abstract syntax because it must deal with ambiguities, redundancies, and errors in the textual form of a document.

If the internal tree is constructed directly from the concrete syntax, it will be larger and more complex than is desirable. Its size makes it unwieldy and occupies much memory; its complexity conceals the structures of the underlying language. However, in order to parse changes to a tree incrementally, the internal tree must be closely related to the tree described by the concrete syntax.

Basing the internal tree on the abstract syntax yields other benefits. First, the abstract syntax is usually closer to the user's notion of the language. This closeness simplifies the mapping between the internal representation of the tree and what the user sees. Second, when the internal representation is shared by several tools in an integrated environment, the simpler form is more desirable. It may also be necessary to shape the internal form to the needs of the other tools rather than dictating the form based on the needs of a parser. Providing internal trees described by the abstract syntax while allowing incremental parsing was one of the major problems in the design of PAN.

Transforming a parse tree to an abstract syntax tree is straightforward. But since the tree-processor needs to be able to incrementally parse changes, the system has to be able to recover the state of the parser (at a given point) from the internal tree. Therefore, the transformation from a parse tree to an abstract syntax tree must be a disciplined one. In general, such transformations are not invertible. But if the parse tree and the abstract syntax tree are "similar" enough, and if a small amount of information about the parse and the particular transformations is retained, invertibility can be assured.

Reference [3] provides a definition for the above notion of "similarity." If two grammars are similar, then the transformations between them can be automatically generated and applied. There is no need to construct a full parse tree; the more abstract tree can be built directly from the parse.

PAN uses three related descriptions of a language's syntax: the concrete syntax, the abstract syntax, and the representation rules. The concrete syntax is a context-free grammar used to build a shift/reduce parser for the language. The abstract syntax describes the internal tree structure; it can be specified using a phylum/operator tree[12]. The representation rules provide the glue between the abstract syntax and the concrete syntax.

Each operator in the abstract syntax has a representation rule showing how to represent the operator in the concrete syntax. The set of representation rules are represented by a context-free grammar. PAN requires that the concrete syntax and the representation rule grammar be similar. For further details, see [3].

PAN uses an incremental LR(1) parser to construct internal trees from the token stream. The incremental parsing algorithm of [10] is extended to transform the parse tree to an abstract syntax tree and to reverse the transformation as necessary. The information required for these transformations will be supplied by the LADLE processor. Tables built by LADLE, together with information retained in the internal tree, allow the system to recapture the state of the parser prior to parsing an altered region.

To implement tree-building, the shift and reduce actions of the LR parser are augmented to consult tables created by the LADLE processor. If a tree-building action is required, an action routine is called. A second stack is used to store intermediate results. Creation of the internal tree becomes a side-effect of parsing. Both the parse stack and the tree-building stack are affected when the parse state is restored prior to reparsing.

Camel[5] is a yacc-compatible[11] LR parser generator written by Robert Corbett as a part of his dissertation research. In the first go-round, the canonical parser provided by camel will be adopted with slight modifications. This code is written in C. The tree-building actions will be LISP functions called by the parser.

Incremental parsing is based on incremental changes to the tokens in the tree. Changes to the token representation can be inferred from changes to a textual representation using an incremental scanner and some information cheaply supplied by the sticky-pointer representation. When changes are made to the token stream directly (as with token- or tree-oriented edit operations), the inference mechanism will be unnecessary.

Error handling will initially be simple: a panic mode mechanism will be implemented which restores the parser and the input stream to some continuable state following the detection of an error. Tokens which are skipped during error recovery will be attached to a distinguished error node that will be put into the internal tree. A similar error recovery method is used in Saga[4]. The tokens involved in the error will be highlighted on the screen, and an error message displayed on the annunciator line of the window.

Internally, trees will be represented using pointers and structures. Each node represents an operator in the abstract syntax. A node contains: a pointer to a descriptor containing fixed information about that node type, a pointer to its parent in the tree, information required for incremental parsing, pointers to its children (if they are internal nodes), and a pointer to the first token in its yield. Tokens, of course, form the leaves of the tree.

3.4.2 Language-Oriented Operations

The algorithms described above provide the desired internal form, a form that can be described by an phylum/operator tree[12]. (Section 4.2.1 contains a brief example.) In an phylum/operator tree, the nodes are called operators, while classes of operators are called phyla. A specification for an phylum/operator tree defines both the operators and the phyla. Language-oriented operations that operate on the phrase structure of a language are defined in terms of the phyla.

The set of phyla is called the *selection hierarchy*. The usefulness of the selection hierarchy is most obvious with respect to commands for traversing the structure. For instance, the *Next-Statement* command will move the locus of attention to the "next" statement in the document as determined by a textual view of next. (Operationally, "next" means "if you can't go in, go right.") Internally, these operations are implemented on nodes.

The selection hierarchy can be used to set a desired level of abstraction for operations. When the selection hierarchy is implicitly used by commands, the current selection level becomes a "mode" of the editor. If used this way, the standard commands such as *Next* or *Delete* could use the current selection level to determine that kind of object is being designated. Such an interface reduces the number of commands (and keystrokes) with which the user would have to cope.

3.4.3 Prettyprinting

Incremental prettyprinting (or unparsing) can be implemented as a tree walk over portions of the internal tree. Each kind of tree node needs to have associated formatting information, and may contain some local state used by the incremental pretty-printer. Formatting information can be embedded in the representation rules, but should be customizable by the user. It will be assumed that prettyprinting does not affect the lexical structure of the document. Thus, prettyprinting can be modeled as a replacement of text regions which does not require reparsing.

3.4.4 Templates

Building a tree using templates resembles the tree-construction steps in a syntax-directed editor. A template-driven form of tree building to PAN is a simple extension of the facilities already present. All that is needed is a way to recognize nonterminal symbols as special tokens during parsing. Appropriate definitions for templates and "nonterminal tokens" can be derived from the LADLE description and added to the description of the lexical analyzer. The templates themselves are defined by the representation rules. (If a handwritten analyzer

is supplied, it would have to be extended to handle these new tokens.) Some consistency checking will be required.

3.5 Semantic Checker

PAN's static-semantic checker is modeled on logic programming. A database of facts about a document is incrementally updated during the editing session. Constraints imposed by the static-semantic specification of a language are checked by evaluating *clauses* against the contents of the database.

Facts constitute the dynamically changing aspects of the database, describing both its structure and its contents. Each fact in the database can be traced to a single assertion in a single construct in a document; thus to a node in the internal tree representation. The incremental parsing algorithm used by PAN unambiguously determines every node directly affected by an alteration in the document. This information can be used to ensure that the database is consistent with the document.

There are three forms of clauses: *axioms*, *constraints*, and *assertions*. An axiom is a clause that is independent of any particular document. Axioms reflect the fundamental definition of a language, such as what it means for two types to be compatible or for an object to be assignable. Constraints are clauses which must be satisfied for the document to be well-formed. Constraints are attached to phrases in the abstract syntax. For instance a typical constraint is that identifiers must be declared before they are used. Assertions add information to the database; they will often be combined with constraints to provide "guarded assertions." A typical assertion would add the fact "The identifier *i* is declared to be a variable of type *integer* in this function."

The basic idea is that when a new structure is added to a document (as in inserting a new declaration in a program), the clauses attached to the new structure will be evaluated. If the clauses cannot be satisfied, an error message will be produced. In terms of attribute grammars, the clauses correspond to attribute functions, and information from particular constructs corresponds to local attribute values. However, the database replaces the flow of attribute values through the syntax tree. Evaluation of clauses by searching the database subsumes attribute propagation.

Several advantages accrue from this model. First, the description of semantic constraints can be written in a form close to that used in the specification of programming languages. Second, the evaluator of the constraints retains control over information flow. Third, because of the way clauses are evaluated, the database and evaluator can support more complex queries about the document. For example, (declared "*X*" ?*type*) might be a clause which returns the type of *X*. The query (declared ?*X* "*integer*") would then return the names of all of the variables declared to be of type *integer*. Such queries can

be made available to the user, perhaps using a syntactically sugared version of the internal language. The database is also available to other portions of the editor for implementing language-oriented commands. Fourth, the general notion of clauses and facts enables the system to gather more information about a program than can be managed in an attribute-grammar or action-routine system. The database resembles a relational-database, and the full power of logic programming can be supported.

To implement this approach effectively, one needs an evaluator, a database, an evaluation strategy, and a means of handling changes to the data base. The last is, perhaps, the most difficult problem. It corresponds in many ways to the problems of truth-maintenance and non-monotonic logic being explored in the artificial intelligence community. Fortunately, the problem presented here is more constrained than the general case.

3.5.1 Evaluation

Evaluation is a process of sequentially satisfying clauses. Each clause is composed of one or more terms. The evaluator takes each term in a clause and attempts to match (unify) the term with either (i) a fact in the database, or (ii) the head of an axiom. If a fact is matched, then the term is satisfied. Otherwise, an axiom has been matched, and it is evaluated with the result determining whether the original term is satisfied. Terms may include "logical variables" that are given values during the process of matching.

There are two ways for a clause to fail in this system: a term could be contradicted, or it may fail because the information is not available. It is not yet clear whether the logic used should be bi-valued or tri-valued.

We assume that the forms of all facts and all clauses are known when the language-definition is processed. This means that new clauses cannot be added during an editing session without reprocessing and reloading some of the language-description tables. Two simplifications come from this assumption. First, it should be easier to handle the incremental changes in the database when all changes are under the control of the system. Second, it may be possible to speed up evaluation by "precompiling" many of the searches and unifications. Techniques for such precompilation will be investigated as part of this research.

3.5.2 The Database

The database underlying most static-semantic specifications is itself logically structured into blocks; in contrast, the database used by Prolog is logically flat. In a programming language, each block in the database corresponds to a "scope" or "name-space" in a program. At any given point in the program, the information available at that point can be found by searching some set of the blocks in the database. Searching is usually order dependent.

It is important to restrict searches through the static-semantic database to those database blocks that are visible at the point in the document where evaluation is taking place. Restricting the search pares down the number of facts which might be matched, and ensures that the look-up will conform to the semantics of the language. Thus, the language for describing static semantics must be able to describe the structure, the contents, and the search rules for the database.

PAN will use the work of Phillip Garrison [8] to define the primitives for constructing and searching the database. In addition, two forms of evaluation are required: one which attempts to satisfy terms by looking only in the current block, and one which performs full search.

The structure of the database should itself be described in the database. This permits a common representation, and the machinery for tracking changes in the contents of the database can be applied to tracking the—necessarily more global—changes in the structure of the database.

3.5.3 Consistency

To maintain consistency and correctness, the system must be able to retract facts and reevaluate all of the clauses whose satisfaction depended on those facts. This implies that each time a fact is used to satisfy a term, a record must be kept. In general, that record will be a pointer from the fact to the node associated with the term in question.

Similarly, if a constraint fails, the system must record that failure as an error in the document. Subsequent changes in the document may allow that constraint to be satisfied. The unsatisfied constraint must be remembered since the change which allows it to be satisfied may do so by indirect action through the database. For instance, adding a declaration to a program may affect a number of constraints throughout the database.

3.6 Extra-lingual Considerations

3.6.1 Comments

Comments are treated by PAN as tokens which are not presented to the parser. A lexical analyser is required to recognize them. Comments will be placed into the internal tree as either *prefix-comment* or *postfix-comment* attachments. (This is the approach taken in the Mentor system[12].) Generally, the placement of comments should reflect the nature of the comment, i.e., what the comment is about.

Attaching comments to tree nodes requires a heuristic for placing comments and facilities for attaching comments to internal tree nodes. The PascAda project[1] developed one such heuristic.

Comments are one of a class of objects that can be modeled as attachments to nodes in an internal representation. Commands for creating and manipulating attachments are part of the planned PAN command set.

3.6.2 Preprocessors

The C preprocessor provides file inclusion, macro processing, and conditional compilation facilities for the C language. Each of those facilities has an impact on a language-based editing system.

Preprocessors are a hold-over from batch compilation. As editors are integrated with compilers and debuggers, preprocessing becomes a curse. A language-based editor must have access to both the input and the results of the preprocessor for it to do its job. Users of a preprocessor need to view and edit the representation before preprocessing, while the language-based components require the results of the expansion. Rather than building-in the entire C preprocessor, one should provide for the generic capabilities used by most languages.

File inclusion is one such capability. Inclusion of one source file in another modifies the database of information used by the semantic checker. It doesn't matter whether the inclusion is an action defined in the language or by a preprocessor; PAN must be able to recognize the action and gather the appropriate information. Right now, the design includes a function which, given a source file name, will gather the information and modify the database appropriately. If necessary, the source file can be loaded into a different buffer and processed normally.

Macro processing is more difficult, because the system needs to support both the unexpanded and expanded view of a macro. (Here we are concerned with the text-oriented macros of a pre-processor rather than the structure-oriented macros of LISP.) One approach is to place the unexpanded text in the text-stream, and the expansion in the token stream (with suitable markings). How to perform expansion is preprocessor-dependent; but any expansion can be modeled as a string-to-string transformation. Macro-expansion does require the ability to define and redefine macros: redefinition of a macro, or definition of a new macro, has to be treated as a global operation on the textual representation.

Conditional compilation requires the creation and maintenance of an "edit-time" environment for determining which sections of a document are semantically and syntactically relevant, and which are merely text. Probably, this can be handled using the facilities for macro expansion (macro definitions must be part of the edit-time environment, too) plus some markers in the token stream.

What is needed is a general capability to bind a command to arbitrary sequences of text or tokens. This binding mechanism must support parameter recognition and replacement.

With such a feature, one can treat preprocessor keywords as functions which define new macros or perform operations; and macros can be expanded by invoking a general expansion function. The bindings to keywords would be performed at language-definition time.

4 Language-Description Language

LADLE is the LANGUAGE-Description Language for PAN. The suite of programs that process a LADLE description and create tables and code for PAN is (in the true UNIX tradition) named *ladle*⁵. From the preceding sections, it should be clear that a LADLE description must have facilities for specifying

- Tokens,
- Concrete syntax,
- Abstract syntax,
- Representation rules,
- Static-semantic rules and constraints, and
- Extra-lingual requirements.

Each of these aspects is described, in turn, below.

The basic strategy of LADLE is to preprocess the description into forms useful to PAN. For lexical and syntactic descriptions, the LADLE description is itself preprocessed before being passed to *lex* or *camel* for table construction. A LADLE processor can therefore provide a higher level of abstraction than the underlying table generator. For example, sequence operators in the concrete syntax are provided in this way; they are transformed into rules acceptable to *camel*.

4.1 Tokens

Tokens are defined by a textual pattern, a symbolic name, a set of attributes, and optionally an integer identifier. The symbolic names must appear in the concrete syntax. A lexical analyzer can be generated by *ladle* using *lex*, or can be supplied.

⁵Where typography fails, context asserts itself.

4.1.1 Specification

If a generator is to be supplied, only the token names, attributes, and integer identifiers must be supplied. If the generator is to be generated by *ladle* the textual patterns must also be specified. Patterns may be specified in one of three ways:

regular-expression	symbolic-name	
regular-expression	{ action }	
string	string	match-type symbolic-name

In the first two cases, if the token has a fixed textual representation, the regular-expression is a simple string. The second form of specification is copied into the *lex* description unchanged. The third form of token specification is used for comments, and other forms that are bracketed constructs. The first and second strings specify the opening and closing brackets. The "match-type" is one of *balanced* or *first*, specifying whether the delimiters nest or not. PAN supplies internal functions for performing the matching.

When a lexical analyzer is generated, the nonterminal symbols appearing in the representation rules are added to the patterns recognized. For instance, if "EXPRESSION" is a nonterminal symbol, the pattern

"<EXPRESSION>" EXPRESSION

is added to the pattern set. The operator "EXPRESSION" can then be displayed as (Expression) and parsed correctly.

4.1.2 Processing

Processing the lexical specification consists of two actions: defining tokens to PAN and creating the lexical analyzer.

4.2 Concrete Syntax, Abstract Syntax, and Representation Rules

Taken together, the concrete syntax, the abstract syntax, and the representation rules for a language define the internal tree structure, its textual representation, and the operations necessary to pass between the two. Prettyprinting information can be specified as a part of the representation rules, but this information does not affect either the structure or the token-representation of the document.

4.2.1 Specification

The concrete syntax is specified using an extended context-free grammar in which sequences of items can be described. Thus, the production for a sequence of declarations can appear as

$$decls \rightarrow \{declaration\}^*$$

rather than as the recursive rules

$$\begin{aligned} decls &\rightarrow \lambda \\ decls &\rightarrow decls\ declaration \end{aligned}$$

Extended rules must appear consistently throughout the concrete syntax, the representation rules, and the abstract syntax: e.g., if the sequence form $decls \rightarrow \{declaration\}^*$ is used in one of the three descriptions, it must be used in all three.

The representation rules are also specified by a context-free grammar, using the same notation as the concrete syntax. Each representation rule is in one-to-one correspondence with an operator in the abstract syntax, so each representation rule can be tagged with the name of the operator that it represents. This correspondence will be carried through to the concrete syntax: each "important" rules of the concrete syntax (see [3]) must be tagged with the operators with which they correspond.

Consistency between the concrete syntax and the representation rules is checked by the ladle processor. To do so, a mapping between the two context-free grammars must be established. The information required for that mapping, namely the "interesting" nonterminals and the proposed rule correspondences, is supplied as a part of the description.

The abstract syntax may be specified as a "phylum/operator" tree[12]. Each node in the tree is an operator. Operators may be grouped into one or more phyla. The operands of an operator are specified by giving a phylum for each operand.

For example, the following is a simple phylum/operator tree for expressions. It has a single phylum, and three operators binary addition, unary negation, and simple variables. The *identifier* operator has no children: it is a leaf and therefore a token.

Phyla:	Expr
Operators:	plus negate identifier
plus:	Expr Expr
negate:	Expr
identifier:	

The phyla defined should be in one-to-one correspondence with the nonterminal symbols of the representation rules. Moreover, the operator specification of the abstract syntax is a simple syntax-directed translation schema based on the representation rules. This constraint will also be checked by ladle.

4.2.2 Processing

Processing the grammars involves:

1. Reading the three descriptions and check for syntactic correctness.
2. Verifying that the token description is consistent with the concrete syntax.
3. Checking that representation rules (as a grammar) is an $\langle H, \varphi, \psi \rangle$ -abstraction of the concrete syntax. Here, H , φ , and ψ are the pieces of information supplied by the mapping information.
4. Checking that the phyla of the abstract syntax are the nonterminal symbols of the representation-rule grammar.
5. Checking that the abstract syntax is a syntax-directed translation of the representation rules.
6. Building the LR parse tables using the concrete syntax and camel.
7. Creating the auxiliary tables used by PAN for tree representation, construction, and expansion.

4.3 Static Semantics

4.3.1 Descriptions

The static-semantic description language is not yet fully designed. Any description will have three parts: a set of axioms that pertain to any document in the language, a set of search rules that specify how to search the database to satisfy a given term, and a set of constraints and assertions that are attached to operators in the abstract syntax.

4.3.2 Processing

Initially, the set of clauses used in the description are completely known by the ladle processor. Currently under investigation are techniques for optimizing and improving the search and update times by precompilation.

4.4 Extra-lingual Information

Not much done on this but for comments. Comments can be defined as either prefix or postfix attachments to operators in the abstract syntax. This section of the description could contain heuristics for attaching comments discovered in parsed text.

If the mechanisms for binding functions to patterns in the text stream are implemented (see section 3.6.2), standard definitions (such as file inclusion) will be found in this section.

5 References

References

- [1] Paul F. Albrecht, Phillip E. Garrison, Susan L. Graham, , Robert Hyerle, Patricia Ip, and Bernd Krieg-Brückner. Source to source translation: Ada to Pascal and Pascal to Ada. In *Proc. of the ACM-SIGPLAN Symp. on the Ada Programming Language*, pages 183-193, December 1980.
- [2] J. E. Archer Jr., R. Conway, and F. B. Schneider. User recovery and reversal in interactive systems. *ACM Trans. on Prog. Languages and Systems*, 6(1):1-19, 1984.
- [3] Robert A. Ballance. Abstract syntax and grammar similarity. 1985. Work in progress.
- [4] Roy H. Campbell and Peter A. Kirsliis. The SAGA project: a system for software development. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 73-80, 1984.
- [5] Robert Paul Corbett. *Static Semantics and Compiler Error Recovery*. Technical Report UCB/CSD 85/251, Electronics Research Lab, College of Engineering, University of California, Berkeley, CA., June 1985.
- [6] Michael J. Fischer and Richard E. Ladner. *Data Structures for Efficient Implementation of Sticky Pointers in Text Editors*. Technical Report 79-06-08, Department of Computer Science, University of Washington, Seattle, Washington, 98195, June 1979.
- [7] E. R. Gansner, J. R. Horgan, D. J. Moore, P. T. Surko, D. E. Swartwout, and J. H. Reppy. Syned—a language-based editor for an interactive programming environment. In *IEEE Spring Compcon '83*, 1983.
- [8] Phillip E. Garrison. Primitives for symbol table access. 1985. Unpublished.
- [9] M. R. Horton. *Design of a Multi-Language Editor with Static Error Detection Capabilities*. PhD thesis, Electronics Research Lab, College of Engineering, University of California, Berkeley, CA., 1981.
- [10] F. Jalili and J. H. Gallier. Building friendly parsers. In *Proc. ACM Ninth Symposium on Principles of Programming Languages*, pages 197-206, 1982.

- [11] Stephen C. Johnson. Yacc: yet another compiler compiler. In *UNIX Programmer's Manual: Supplementary Documents*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California, 1984.
- [12] G. Kahn, B. Lang, B. Mélése, and E. Morcos. Metal: a formalism for specifying formalisms. *Science of Programming*, 3:151-188, 1983.
- [13] B. W. Lampson. *Bravo Users Manual*. Xerox Palo Alto Research Center, Palo Alto, 1978.
- [14] M. E. Lesk and E. Schmidt. Lex—a lexical analyzer generator. In *UNIX Programmer's Manual: Supplementary Documents*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California, 1984.
- [15] R. Medina-Mora and P. H. Feiler. An incremental programming environment. *IEEE Trans. on Software Engineering*, SE-7(5):472-481, 1981.
- [16] David Notkin. The GANDALF system. *Journal of Systems and Software*, 5(2):91-106, May 1985.
- [17] T. Reps. *Generating Language-Based Environments*. Technical Report TR 82-514, Department of Computer Science, Cornell University, Ithaca, August 1982.
- [18] R. M. Stallman. *EMACS Manual for TWENEX Users*. Technical Report AI Memo 555, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, 1980.
- [19] R. M. Stallman. EMACS, the extensible, customizable, self-documenting display editor. In *Proc. of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 147-156, 1981.
- [20] Jeffrey S. Vitter. USer: a new framework for undoing. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 168-176, 1984.