DESIGN OF A MULTI-LANGUAGE EDITOR WITH STATIC ERROR DETECTION CAPABILITIES

by

Mark R. Horton

Memorandum No. UCB/ERL M81/53

July 1981

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ABSTRACT

DESIGN OF A MULTI-LANGUAGE EDITOR WITH STATIC ERROR DETECTION CAPABILITIES

Mark R. Horton

Programmers who prepare programs interactively usually edit them in terms of text. Those few who use source language editors are restricted to a single language for which they have a tree structured editor.

The **Babel** set of programming tools is described. Babel has knowledge of the syntax and semantics of many languages. A language description language is used to make it easy to add a new language to the collection. Programs are represented as generalized parse trees.

The Babel system includes an editor that presents a text-like interface to the user. Commands utilizing the tree structure are possible, but need not be used. The user can be unaware of the underlying tree structure. The editor incrementally checks the syntax and semantics of the program and immediately notifies the user of errors. Human factors involving temporarily incorrect programs, program formatting, and responsiveness of the editor are addressed.

A prototype implementation has been built. Its design and characteristics are described.

Acknowledgments

I am indebted the members of my committee for their encouragement and assistance in this project, especially Professor Susan L. Graham, for always encouraging me to improve on what I had.

The excellent computing environment, brought about by many graduate students at Berkeley, was an invaluable aid in both implementing the programs making up this research, and for the production of this dissertation.

I would like to thank Eric Allman, Mike Deering, Dan Halbert, Robert Henry, Peter Kessler, Kirk McKusick, and especially Dave Menicosy, who proofread earlier drafts of this dissertation and made many valuable comments.

I also would like to thank the many people who tried out my system while it was still under development, finding bugs and showing me how people would use it.

Most importantly, a special word of thanks to my wife, Karen, who has been a constant source of support and help throughout my years as a graduate student, for many hours of proofreading and editing of this dissertation, and for always being there when I needed her.

Work reported herein was supported in part by the National Science Foundation under Grants MCS-8005144 and MCS 74-07644 A04, and by Defense Advanced Research Projects Agency (DoD) ARPA Order No. 4031, Monitored by Naval Electronics Systems Command Under Contract No. N00039-80-K-0649.

Table of Contents

CHAPTER 1

Introduction

1.1. The Problem

With current programming tools, the programmer spends most of his time in an *edit-compile-debug* cycle. While developing or modifying a program, it is typical to have to make several passes through the cycle just to "get the semicolons right" before real debugging of the program can begin. The cycle usually includes the tedious process of getting a list of error messages, reentering the editor, fixing a few bugs, recompiling, and repeating for the next batch of errors. Much time is spent shifting from one phase to the next.

An approach receiving attention recently is the use of a *language editor.* A language editor is an editing tool, replacing a text editor, which has access to information about the language being edited. To make an efficient implementation possible, language editors usually do not edit text directly, instead using a tree representation. Language editors have a number of potential advantages over text editors.

(a) Errors can be detected, or even prevented outright, as the program is entered. There is no need for the user to compile the program to get the syntax checked, or to make sure that all variables are declared.

(b) Program-oriented structure operations are easier to implement correctly than in a text editor. Although many good text editors have commands to reference such structures as *statements, sentences,* and text containing *balanced parentheses,* such commands are usually implemented by searching for a known, language dependent *delimiter string* such as "**end**" or ")". They are usually fooled by comments and strings containing the delimiter string being searched for.

(c) The editor can do a more intelligent job of elision. Most text editors elide (i.e., do not display) lines that are above the top of the screen or below the bottom. If the programmer wants to look at the statements that surround a large loop, there is often no way to get both ends of the loop on the screen at the same time. When the tree structure is known it becomes possible to manually [Teitelbaum 1979] or automatically [Alberga 1979] elide part of the program, replacing an arbitrarily large block of code on the screen with an elision indication such as "...".

(d) Keeping programs in tree form can be useful in other parts of the system. There is no reason to convert from tree to text and back upon each entry into or exit from the editor. If the compiler works directly from the tree, its job is simplified (and sped up) because it need not scan and parse the program. It is possible to edit the tree *in place*, avoiding the customary copy when entering and leaving the editor.

(e) The editor can mark the parts of the tree that have been changed. This provides an excellent environment for incremental compilation and for keeping histories.

(f) If a suitable tree representation is chosen, additional semantic information that the compiler, editor, and other tools ignore can be stored in the tree. This allows a symbol table, generated assembly language code, and other semantic observations to be kept in the tree.

Most of the existing language editors are for a specific language. To port the editor to a different language, substantial modification to the program would be necessary. Those few that handle more than one language have other problems. The editor described in [Wilcox 1976] is written in an assembly level language, and requires substantial amounts of reprocessing for cursor motion. The proposal of [Feiler 1980] use a formalism which is not self-contained: external routines must be written and linked into the editor. Being able to easily add a new language to the set available to the editor is a very desirable property.

The interface presented by many language editors to their users is based on the tree being edited. The user must thoroughly understand the tree structure. This limits the user community of the editor to people who are able to understand the tree structure, usually excluding non-programmers. Since the user of the system to be described here sees the text of the program on his terminal, and since text is straightforward to understand, an interface which presents the user with the text is likely to be more easily learned than a tree interface.

There are advantages to a text interface for experienced programmers, too. [Stallman 1978] cites a number of advantages of text editors over the tree oriented language editors that existed at the time. Typical of the advantages cited is that a text editor allows the programmer to format his program, including comments, as he sees fit rather than being forced to use a standard layout. Also, a text editor allows the program to be temporarily incorrect during multiple command operations such as adding a pair of parentheses, whereas a tree interface usually prevents syntax errors from occurring, requiring a less natural editing style. Finally, a text interface enables the user to have the same familiar tool for editing programs that he uses for editing other kinds of text.

Both types of systems have their own advantages. The best of all worlds would be a hybrid approach, having a user interface with all the simplicity and power of a text editor, yet able to perform operations requiring a tree, such as checking for errors, or correctly finding structurally defined portions of the tree. Such an approach is taken here.

1.2. The Properties of a Solution

1.2.1. Information Required

There are several kinds of language information a language editor requires. It must have access to information describing the syntax of the language in order to build a tree to work from, and to check for syntax errors. It needs the lexical structure of the language if the user is to type text. (The template oriented approach of [Feiler 1980] does not need the lexical information, since it insures that the user never types in tokens, other than identifiers and constants prompted for by the system.) If the system is to check semantics, it must have the semantic rules of the language.

1.2.2. Table Driven

The notation for this information should be high level, self-contained, portable, and powerful enough to describe most modern programming languages. It should not contain information irrelevant to the language being described. In the implementation, the language dependent information should be all kept in one place, rather than scattered about, embedded in various programs. A definition for a language should be free of machine dependencies, so that it can quickly be ported to another environment.

A notation for description of languages should be completely self contained, so that tables generated from the description can be the complete source of language dependent information. This provides for a cleaner, more portable design. It also allows a user to substitute his own version of a language, encouraging improvements to language dependent information such as error detection and language dependent macros.

1.2.3. Erroneous Entry Possibilities

One important advantage to current text editors is that they permit errors. While detection of errors by the editor is a valuable programmer aid, in a text entry environment it is unreasonable for the system to refuse to accept the change until it is entered correctly, since the change may be mostly correct. Many users make multiple-change corrections that go through illegal states before entering a final, correct state. (Examples of such multiple stage changes are moving a parenthesis, or adding a **begin-end** pair around a block of code).

The tree structure underlying the implementation must behave cleanly in the presense of errors. If a text-editor interface is to be presented to the user, programs that are syntactically incorrect must be displayed exactly as typed, so that the user can correct them as with a text editor. If the program is in an error state and a command is typed causing the program to become correct, the editor must form a correct tree.

1.2.4. Well Engineered Error Messages

Another important consideration, if errors are to be detected, is how to make the user aware of the errors. Preventing errors outright is undesirable for reasons discussed above. We feel that restricting the cursor position to the area before the first error, as done in [Wilcox 1976] and [Morris 1981] is overly restrictive. Producing an error message on the terminal screen for each error encountered is likely to annoy the user, especially when the user is knowingly going through an incorrect state to get to a correct state; it would also probably produce voluminous output, taking time and valuable space on the screen. Notification of errors must be easily noticed by the programmer, yet low-key enough to not get in his way.

1.2.5. High Degree of Incrementality

An editing system which checks syntax and semantics must do considerably more work than a text editor. If a small change is made to the program, a very incremental system will recompute only a small amount of local information. A high degree of incrementality is important.

1.2.6. Shared Symbol Tables

An implementation that checks semantics will have to keep a symbol table, to check for proper use of symbols. Most implementations that use a formal semantic model are based on attribute grammars [Knuth 1968]. In such an implementation, attributes are normally copied around the tree liberally. Since a symbol table would be a very large attribute, it is important not have multiple copies of symbol tables. The time and space requirements of making such copies would be excessive. Some provision must be made for sharing the symbol tables, while allowing for correct updating as the user makes changes.

One serious problem with the usual implementation of a shared symbol table (a pointer to a static table) is the deletion problem: what to do if a declaration is deleted, or the name is changed. Most current compiler technology is based on the assumption that the compiler starts with nothing and builds up by adding one item (declaration, etc) at a time. There is no provision for deletions.

The problem is even worse in the case of an editor based on an incremental parser. Not only must the editor correctly handle deletions, but even determining that a deletion has taken place is difficult.

1.3. The Babel System

This dissertation describes a set of tools called **Babel**. The most important of these tools is the editor **be**. The interface presented the user by **be** is that of a text editor. The implementation, however, keeps a parse tree representation of the program rather than a text representation. This dissertation shows how to build a text interface as the front end of a tree representation. The system checks for syntax and semantic errors, and reports them unobtrusively to the user. Structure oriented commands can be implemented correctly by making use of the tree structure.

The Babel system includes incremental evaluators for lexical, syntactic, and semantic information. A table driven scanner is included, which can be used for subsections of the program alone. An incremental parser for context free grammars, based on that described in [Ghezzi 1979] checks the syntax after each line of input. An incremental attribute evaluator, based on that in [Reps 1981] checks semantics as the program is changed. These checks can be partially or completely turned off by the user, to improve response time, without affecting the text editor interface. (An explicit **check** command can be used, in this mode, to check for errors.)

Context free grammars are an almost universally accepted notation for specification of syntax, and easily implemented subsets are more than powerful enough for most current programming languages. Regular expressions are often used for lexical information, and have moderate power. (They cannot handle lexically unusual languages such as FORTRAN, with its conventions for columns 1, 6, and 7, and its use of blanks.) Attribute grammars [Knuth 1968] have the power to express the static semantics of most programming languages, and given a powerful enough notation for attribute evaluations, can handle the static semantics of any programming language.

The Babel system uses a completely self contained notation for describing languages called **Language Description Language** (LDL). LDL is based on the three formalisms described above. LDL is a completely self-contained language which describes a large class of languages, including LDL itself.

An expert *language implementor* prepares an LDL description for a language, and runs it through a preprocessor to produce a set of tables. The editor uses these tables as a language-dependent knowledge base.

LDL is very high level. The lexical and syntactic portions of a new language can be brought up in a few hours, given a grammar for the language. Specification of semantics takes considerably longer, but still requires less time than it would to write the front end of a compiler,[1] because of the high level of the attribute grammar notation. LDL encourages highly portable language descriptions. Such descriptions would also be of use in the production of portable compilers.

Syntax errors are handled in Babel by building a tree which corresponds exactly to the incorrect program typed by the user. Structural information near the root of the tree is lost, but is quickly regained when the program is corrected. When this incorrect tree is

[1]A syntax-only description of Lisp was developed in under an hour. A similar description of Ada worked after one evening. A description of Pascal powerful enough to detect undeclared variables and types took about three days.

displayed, the same text the user typed will appear. Most importantly, when another change (producing either a correct program or an incorrect one) is entered, the parser will treat the incorrect tree the same as a correct tree, producing a correct tree if the program has been corrected, and reparsing little of the tree.

Error messages of all kinds (lexical, syntactic, semantic) can be attached to any node of the tree. Nodes with error messages will be highlighted[2] on the screen, pointing out the error and its location quickly and quietly. (If the node is a nonterminal, its descendents which are tokens will be highlighted.) The user can determine the cause of an error by positioning the focus to the highlighted node and asking the editor for the text of the error message.

Symbol tables are shared by building a symbol table one entry at a time, using a linear linked list. This method arranges that incomplete versions of the symbol table are still valid, and can be used for the construction of changed versions of the symbol table. The deletion problem is handled by rebuilding the remainder of the symbol table from the point of the change. (The symbol table entries themselves do not need to be recomputed, only the parent nodes that link the entries together.) A hash table at the top of the list provides fast lookup once the table is built. A symbol table building block is provided which is flexible enough to be used not only for block structured languages, but for languages with other scoping rules as well; yet the building block is high level and frees the language implementor from the details of constructing an efficient symbol table.

1.4. Results

The approach outlined here has been implemented to yield a working editor. Students have been voluntarily using the editor, which has kept statistics showing the amount of

[2]It is assumed that the user has a terminal with some form of highlighting, such as bold, reverse video, blinking, or underlining. The screen package that our implementation of be uses [Arnold 1980] ignores highlighting requests on terminals that cannot highlight, but could be modified to bracket highlighted text in some way, such as putting angle brackets around <<the highlighted text>>.

work it had to do. In addition, an editing session on one particular Pascal program was timed using various text editors and using be with various amounts of checking. For most cases, a high degree of incrementality has been attained. Even so, the prototype editor is several times slower than local production text editors, depending on the amount of checking done and the nature of the changes.

Syntactic checking is shown to be within reasonable efficiency constraints, being roughly five times slower than **vi** [Joy 1980]. Semantic checking is somewhat slower, losing a factor of about 15 when an executable statement is changed, and a much larger factor (depending on the size of the program) when a declaration is changed. Automatic checking can be turned off for the duration of some sequence of commands, or permanently, with checks only by explicit user request, speeding up the process at the expense of some checking. It seems likely that a more careful implementation could improve the ratios considerably.

The remaining chapters of this document describe and motivate the Babel system, and compare it to other systems. Chapter 2 describes some other language editors and compares them to Babel. Chapter 3 defines notation used in the other chapters. Chapter 4 shows the user interface, with an example session, and discusses the design of the particular command language chosen. Chapter 5 describes the implementation of the editor. Chapter 6 describes LDL and the implementation of the LDL processor. Chapter 7 contains some performance results, and lists some directions for future research.

CHAPTER 2

Previous Work

2.1. Introduction

This chapter discusses some of the research done by other researchers in the area of language editors. In addition to several earlier language oriented editing systems [Teitelman 1978, Donzeau-Gouge 1980, Teitelbaum 1979b, Alberga 1979, Yonke 1975, Feiler 1980, Wilcox 1976], related research is discussed which develops techniques used and extended in this dissertation for formal language definition [Knuth 1968, Geigerich 1979], incremental parsing [Ghezzi 1979], and incremental attribute analysis [Demers 1981, Reps 1981]. A comparison is made between the other editing systems and Babel on each of several issues, including the language(s) handled, lexical issues, comments, parsing method, semantics, prettyprinting, editing style, incomplete or incorrect programs, whether the system is screen-oriented, to what extent the system has been used, methods of elision of part of the program for display purposes, preprocessors, and execution environments.

2.2. Teitelman

The Interlisp system [Teitelman 1978] is probably the most well known of the systems containing *Lisp editors*, that is, *tree editors* that edit Lisp programs. Since Lisp programs are intrinsically tree structured, with a very simple syntax, it is significantly simpler to build an editor for Lisp than for most other languages. Such editors are common in Lisp environments, although it has been argued [Stallman 1978] that text editors are superior to tree editors even for Lisp. Interlisp has been very widely used since about 1970 on Tenex and TOPS-20 systems.

Tree editors can be characterized by their internal representation, which is a tree that represents the syntactic structure of the program, rather than the characters, lines, or pages that text editors use to represent program text. Interlisp edits Lisp S-expressions. (Since S-expressions are so simple, this is easy to do for Lisp.) Interlisp does not know any semantics, nor does it know any of the syntactic forms embedded in S-expressions commonly used in Lisp (such as **cond**, **prog**, or **lambda**). It is simply an S-expression editor, which allows it to be used to edit data as well as program text.

This system has been tuned over the years to be convenient to use. It has a rich set of tree editing commands, including "parenthesis changing" commands which allow the user to think in terms of adding, removing, or moving a parenthesis while actually making a change to the internal tree. A "Do What I Mean" (DWIM) feature is called when an error is discovered. DWIM has been tuned to recognize common mistakes and make a correction that will often be what the user intended.

Interlisp is a Lisp *programming environment* That is, it is a single large program the user runs continuously while developing a program in a particular language. There are many other programming environments, most commonly for the languages Lisp, Basic, and APL. More recently, programming environments for the language Ada have received considerable attention.

Probably the most interesting feature of Interlisp and its editor is the history list. Each elementary change made by the user is recorded in a history list, allowing changes to be undone and redone at will. After making a mistake, the faulty command can be undone with the "undo" command without having to think of the command or sequence of commands necessary to undo it manually. The history list allows multiple commands to be undone, all the way back to the beginning of the editing session. The user need not fear losing any work because all commands which are undone can be redone just as easily with the "redo" command.

2.3. Donzeau-Gouge et.al.

The Mentor system [Donzeau-Gouge 1980] is a Pascal tree editor. It knows the syntax for the entire Pascal language, and can be programmed to handle the semantics. It is teletype oriented, not using any CRT screen capabilities.

Mentor allows the user to think of the program as either a tree or text. There are many tree oriented printing commands. When printing, a root and the maximum tree depth to print can be specified. This causes the subtree with the specified root to be printed, eliding (e.g. printing as "..." or "&") all portions beyond the specified depth. As is the case with most tree editors, prettyprinting is done on the fly.

Program text is entered to Mentor by telling it that one wants to input a particular nonterminal at a particular point. After checking to see if that nonterminal is legal at that point, Mentor prompts the user for that nonterminal, which is parsed in a goal-driven manner. Input text which can be parsed to exactly that nonterminal must then be entered. No provision is made for unstructured input (input that is not a subtree but rather a group of neighboring branches, such as "+3" or "end; begin") or incorrect input. Thus, the editor can guarantee that the program is both complete and correct at all times.

Mentor commands are very powerful and concise, much like those of [DEC 1972]. A strong pattern matching capability is defined, utilizing the tree structure. Changes to existing programs that do not involve new text are made with a rather low level set of operations, that correspond roughly to the primitive operations that can be made on trees (prune subtree, replace leaf, exchange two children, etc.).

2.4. Teitelbaum

The Cornell Program Synthesizer [Teitelbaum 1979b, Teitelbaum 1979a] is a tree editor for a subset of PL/I called PL/CS. It runs on a Terak personal microcomputer, and is in use in introductory programming classes at Cornell. A more recent version runs under the UNIX[1] operating system on a VAX[2].

The Synthesizer is screen-oriented and makes full use of the hardware available. It knows both the syntax and semantics of PL/CS, with emphasis on the syntax. When the user makes a semantic error, such as using an undeclared variable, the incorrect use is highlighted on the screen. Syntax errors are not possible.

Input is not textual, but rather consists of a sequence of template building commands. Each production in the grammar has a corresponding command to create a template from that production. For example, the command **.ite** will create an *if-then-else* node at the current location in the tree, leaving the *condition*, *then part*, and *else part* as nonterminals to be filled in later, and placing the focus at the *condition*. This program construction stops at the expression level, allowing expressions to be typed in and parsed normally.

Since the user does not type in the program as it appears on paper, but rather builds a tree, the tree always corresponds to a syntactically correct PL/CS program. There is no way to represent a syntactically incorrect program, and hence syntax errors cannot occur. (Incorrect expressions are rejected immediately.) A program can be *incomplete but not incorrect*, that is, there can be nonterminals which have not been expanded, but no syntax errors are possible.

Such an editor forces a totally different editing style on the user who is used to a text editor. Programs cannot easily be entered from a handwritten listing, but must be constructed a node at a time. Changes cannot go through intermediate states with syntax errors. (Temporary introduction of errors is a widely used technique with text editors. For example, moving a sequence of statements inside a **begin-end** block is usually done by deleting the **end** and adding another **end** after the affected statements.) In the synthesizer,

[1]UNIX is a trademark of Bell Laboratories.

[2]VAX is a trademark of Digital Equipment Corporation.

many changes which make sense textually cannot be done with any straightforward command. For example, using a text editor, changing an **if** statement to a **while** statement in a language such as C can be done by a one word substitution, since the two forms differ only in the keyword used. The synthesizer would require the body to be deleted into a logical register, the **if** header deleted, a new **while** header typed in, and the body restored from the register underneath the **while** node.

There are two forms of elision in the Synthesizer. The more straightforward method is to elide anything more than a certain number of lines away from the focus. (This is the approach taken by most screen text editors.) The user can also manually elide subtrees by positioning the focus at the root of the subtree and pressing the elide key, causing that subtree not to be displayed. A second press undoes the elision.

An unusual aspect of Teitelbaum's work is the way it handles comments. Comments in PL/I and in PL/CS begin with "/*" and end with "*/", and may appear between any two tokens. The synthesizer, however, views comments as just another syntactic form:

```
<stmt> ::= /* <text> */
                  <stmt>
```

The <text> is treated like the text of a string, and may be edited using operations similar to those of a text editor. The comment may not be more than one line long. Multiple single line comments may appear in sequence before procedures.

The <stmt> on the right hand side is viewed as a child of the <stmt> on the left hand side, and indented accordingly by the prettyprinter. A similar production exists for declarations, and a comment is *required* by the grammar at the beginning of each procedure. A benefit from this structure is that the right hand statement can be elided, effectively replacing a bulky statement on the screen with a one line description of what it does. This rewards users for using short, descriptive comments.

2.5. Alberga, et. al.

The LISPEDIT system [Alberga 1979] is a complete Lisp programming environment. It is quite different from a conventional Lisp environment such as Interlisp. Rather then merely giving the user tree manipulation commands, the user sees the program largely as text. It is screen-oriented, and runs on IBM hardware.

The system uses an LALR parser and a grammar including syntactic forms such as **cond**, **for**, and **prog**. To add to a program, text is typed in, a line at a time, and parsed incrementally. Any expected right parentheses are automatically added by the parser, and right parentheses which are typed in are interpreted as commands to move up in the tree, thus allowing the user to type in the program as it is written on paper.

The display is prettyprinted according to built-in Lisp prettyprinting rules and a priority elision algorithm which favors the printing of branches in the tree that are closer to the focus. Every time the tree is changed or the focus is moved, the display is prettyprinted. This automatic elision and reprettyprinting cannot be turned off or controlled by the user, which makes it suitable primarily to high speed environments such as the 50K Baud IBM environment. Comments are viewed as footnotes to a token. The prettyprinter prints them wherever there is space: to the right, on the previous line, or at the bottom of the screen with a footnote-style marker.

Erroneous syntax is possible in the tree. The offending tokens are made into error nodes, which are treated similarly to comment nodes, and placed in the tree to the side of the nearest correct node. When the next change is made, the error nodes will be rescanned and reparsed to produce another chance at a correct parse. Allowing errors to be in the tree gives the user much of the power of a text editor that is missing in many tree editors.

LISPEDIT knows the full semantics of the IBM dialect of Lisp. It detects, for example, undeclared variables, and has a convenient command to declare such a variable without moving the focus. An interpreter and compiler are also built into the system, along with a pictorial execution mode for debugging (showing the code being executed as it is run). LISPEDIT is a programming tool intended for use with large software projects. It has been used to develop itself and other large Lisp projects at IBM.

2.6. Yonke

Yonke's 1975 Ph.D. dissertation [Yonke 1975] describes a true language independent tree editor, called PCM (Program Constructor and Modifier). The editor uses a language description formalism to describe the syntax and a limited amount of the semantics of the language in question.

PCM is not screen-oriented. It was influenced by Interlisp, in which it was implemented. It gives the user a set of tree-changing commands such as *insert before*, *insert after*, *replace*, and *delete*, which are appropriate for the general trees in the representation. The input is typed by the user as it is written on paper, and parsed by a goal-driven mechanism which appears to the user to be similar to that in Mentor.

An unusual wrinkle of PCM is that it does not use the traditional context free grammar for syntax definition. Instead, it uses a high level formalism which resembles SNOBOL patterns, with *ordered sequence*, *alternating sequence*, *bracketed sequence*, and *repeating sequence* as the building blocks. Bracketed sequences are not strictly necessary, since they are a special case of ordered sequences, but were included to facilitate error recovery.

This representation is much easier for the language implementor and the user to deal with than an LR parse tree (which relies on recursion and chain productions that the user would not ordinarily want). It is not nearly so well understood as context free grammars, and is vulnerable to such problems as the order of productions being significant, and the need to back up.

The semantics developed by Yonke are limited but useful. PCM builds a symbol table suitable for a block structured language and detects undeclared variables. It also attempts to discover variables which are used before they are set. Since this information cannot always be determined at compile time, the diagnostic given is a warning. The semantics are specified in the formal description as notes attached to productions. For example, "(NEW ACCESS NODE)", indicates entry to a new scope.

While Yonke went far toward forming a language independent editor, he did not address lexical issues or comments at all. Tokens are scanned by the Lisp parser, and comments are not allowed in programs edited by PCM.

2.7. Wilcox et.al.

The CAPS system [Wilcox 1976] is an editor/interpreter used for small subsets of several programming languages, including FORTRAN, PL/I, and COBOL. The system checks the syntax and semantics of the program, as it is typed in, after each keystroke. When an error is found, the system leads the user through a series of possible corrections, based on the underlying grammar. CAPS is not a tree editor. There is no tree representation kept, only a text representation, so the interface presented is that of a text editor.

A small degree of incrementality is achieved by saving the state of the translator at intervals and recompiling from the last saved state before the change. The editor maintains the invariant that a compilation is in progress, which has progressed to the location of the cursor. Since the states are saved for every character close to the cursor, and further apart toward the beginning of the program, and the recompilation stops at the cursor, only a small amount of recompilation is typically done for a local change. Moving the cursor forward requires recompilation of areas moved over. Moving the cursor backward causes information to be thrown away. The disadvantage to this approach is that when leaving the editor, or making a large jump forward, the entire remainder of the program must be recompiled.

CAPS uses a state transition mechanism to handle the scanner, with power comparable to that of the Babel scanner. The grammar is hand-coded using recursive descent in an assembly-level language, which also updates the symbol table. No mention is made of the structure of the symbol table.

In order to keep the tables small, the vector of previously saved states holds only the changes from the previous state. Because of the extremely small limits imposed by the computer on which CAPS runs, only very small programs could be handled.

Since CAPS asserts that a correct compilation has occurred to the left of the cursor, it does not allow the user to move the cursor to the right of the first error in the program. Any keystrokes which would go past the error are ignored.

Wilcox and his group tested CAPS on introductory programming students, and found the response to be much too slow to be usable in practice. Since the computer system often had 500 users logged in, this could have been due to the heavy load on the system. It also could have been because of the lack of full incrementality.

2.8. Feiler and Medina-Mora

Feiler's and Medina-Mora's work at Carnegie Mellon University [Feiler 1980] is a recent investigation involving a multi language tree editor, called IPE (incremental programming environment). The editor is in many ways similar to [Teitelbaum 1979a].

The user program is entered using a constructive language. Teitelbaum stops at the expression level and instead parses; IPE carries the construction method to all levels of the tree. The editing operations are tree oriented. The user program can at any time be incomplete, but never incorrect.

IPE is an *editor generator.* The system is not merely an editor, but also includes an incremental (at the procedure level) compiler and loader. A grammar is entered for a given language, along with a set of externally compiled procedures written in any convenient implementation language. After leaving a procedure, a routine is called to translate the procedure and check for errors.

This system, like the Teitelbaum system, places an emphasis on the tree structure. There is no scanner or parser. The only terminal symbols that are input are those which are not keywords, e.g. identifiers and constants. These are explicitly marked as such by the user. The relationship with text is through an *unparser* that displays text to the user. The unparser includes a set of rules for prettyprinting.

2.9. Eickel et al

The MUG2 compiler generating system [Geigerich 1979] provides an excellent environment for the construction of multi-pass compilers. A high level description of the straightforward aspects of a language (the lexical and syntactic aspects, and portions of the semantics) is input to a preprocessor. This produces a program that can be link edited with external semantic routines provided by the language implementor to produce a compiler.

The language description consists of a lexical description, an LALR(1) or LL(1) grammar specifying the syntax, a string-to-tree grammar used to produce an abstract program tree, an attribute grammar description of the semantics, tree optimization rules, and templates for intermediate code generation.

The lexical specification is regular expression based, and similar in power to that of Lex [Lesk 1979], but the specifications are somewhat more readable. The context free grammar is conventional and merged with the string-to-tree grammar. A reduction by a given production builds a tree section according to the string-to-tree rule corresponding to that production. The attribute grammar and later phases work with the abstract tree instead of the concrete parse tree (which is never constructed), and call external action routines written by the language implementor in Pascal or PP440. (PP440 is a language specific to their particular computer.)

The MUG2 system makes compiler construction much easier and faster than conventional environments that have only scanner and parser generators. It automates many tedious and error-prone portions of the compiler (including information propagation and stack handling) while giving the language implementor the power of an existing programming language for the less straightforward work. It also encourages modularity by making the implementor communicate with the rest of the compiler through parameters, rather than global variables.

However, Eickel and his group have made several design decisions which, while correct in the context of a compiler generator, are not compatible with an interactive, incremental, entirely table driven environment such as Babel. For these reasons, nothing in Babel has been taken directly from MUG2, although many aspects are similar.

The MUG2 system, since it is geared to compiler production, throws away information from phase to phase that is not needed in later phases. Information thrown away includes lexical information such as white space and comments, and the parse tree. In order to reconstruct the source from the internal representation, and to parse incrementally, this information must be present in the internal representation.

Another problem, for our purposes, is the use of externally compiled routines. MUG2 is a *compiler generator* which produces a separate compiler for each language. While an *editor generator* is a possibility, we rejected it in favor of a completely table driven editor, to increase the portability of the language descriptions, and in order to produce a cleaner design. Another possibility would be to load binary absolute or relocatable code dynamically into a running program. This approach has even more severe implementation and debugging problems. Instead, Babel uses tables generated from a completely self-contained, formal description notation based on Lisp.

2.10. Ghezzi and Mandrioli

The incremental parsing technique used in our system is basically that in [Ghezzi 1979]. Some simplifications are made which dramatically cut down on the bookkeeping overhead without changing the overall character of the method.

The Ghezzi method starts with an LR parse tree for a string $w = x\bar{x}z\bar{y}y$, and a new string $\bar{z}$ to replace z. ($\bar{x}$ and $\bar{y}$ are one token of context in each direction.) The strings of symbols (nonterminals and terminals) α and β, where $\alpha \rightarrow^{*} x$ and $\beta \rightarrow^{*} y$, are determined from a set of so-called "threads" the method maintains. (In the terminology of this paper, α is the *left firewall*, and β is the *right firewall*.) Then the string of symbols $\alpha\overline{xzy}\beta$ is parsed using an almost conventional LR parser, with conventionally generated LR tables from a slightly modified grammar (all nonterminals are also terminals). The modification to the parser arranges for the reduce function to build a new parse tree node for the new nonterminal, as the parent of the subtrees from the semantic stack corresponding to the symbols on the right hand side of the production, and to push the new node on the semantic stack.

Ghezzi's method works on the class LR(1) $\cap$ RL(1), which encompasses most practical LR grammars. It also works with an LALR(1) parser on LALR(1) $\cap$ RL(1) grammars, and with SLR(1) on SLR(1) $\cap$ RL(1) grammars. It can be implemented directly using YACC [Johnson 1978].

The simplifications we have made involve the threads and the LALR tables. Ghezzi's paper describes two "threads" kept in each node, which always point to the next node in the firewall in both the forward and backward directions. (Ghezzi's "threads" are not conventional threads, since they take up two extra fields in each node, rather than using otherwise null fields as conventional threads [Knuth 1973] would.) The forward "thread" of a node is defined to reference either (a) its left sibling (if one exists), (b) the left sibling of the closest ancestor that has a left sibling (if such an ancestor exists), or (c) a special "null" node. The backwards thread is symmetrically defined. (Ghezzi views forward (left) and backward (right) as toward the front and back of the tree, respectively.)

Ghezzi does not assume a node contains pointers to its parent, left sibling, and right sibling, although either the left or right sibling would presumably be part of the tree representation. If such node pointers are part of the tree structure, as in Babel, the "threads", as defined above, are easy and fast to construct when needed and need not be maintained. Since either the left or right sibling will already be present, at most two extra fields are needed, requiring no more space than the two "threads" in Ghezzi's method. Such extra fields are often useful for other purposes, so this may actually result in a space savings. The major benefit, however, comes from not having to maintain the "threads"; saving over half of the code described in Ghezzi's paper.

The other simplification is to use the unmodified tables from the unmodified grammar. Instead, the parser is slightly changed to allow nonterminals to appear in the input, simulating the last part of a reduction that leads to that nonterminal (e.g. application of the goto table).

A later paper [Ghezzi 1980] improves on this method by finding a nonterminal, *A*, such that the entire changed area is below *A* in the tree. None of the tree structure above *A* need be reparsed or rebuilt. (In effect, the method described in [Ghezzi 1979] uses the root of the tree as *A*.)

2.11. Research on Attribute Grammars

There are a number of formal techniques available to describe semantics [Knuth 1968, Knuth 1973, Ledgard 1977, Wegner 1972, van Wijngaarden 1975]. All of these techniques have serious drawbacks, either from lack of expressive power, or necessary inefficiency of any implementation.

Several recent implementations based on formal semantic descriptions have used the notation of *attribute grammars* [Knuth 1968, Knuth 1973]. While this notation is lacking in the power to describe runtime semantics, it is well suited to implementations because it is possible to produce a system with only a small loss in efficiency over a hand-coded system.[3]

[3]According to Harald Ganzinger [Ganzinger 1980], the best attribute grammar based systems lose only a factor of 2 or 3 to hand-coded systems.

The attribute grammar notation is based on assigning a fixed number of *attributes* to each token and nonterminal in the grammar. Each attribute has an *evaluation function* used to compute its value. The only arguments these functions are allowed are other attributes of symbols in the same production. Since each nonterminal instance in a parse tree will appear in two productions (on the left hand side of one, and the right hand side of another, corresponding to the productions below and above it in the tree) it is possible for any given attribute to be evaluated in either production. For consistency, attributes are required to be either *inherited* (evaluated in the production above) or *synthesized* (evaluated in the production below). Intuitively, inherited attributes are passed down the tree, while synthesized attributes are passed up.

The area of attribute grammars is currently a very popular research topic, with a large number of papers published in the last decade. (See [Raiha 1980] for a comprehensive bibliography.) The field is still only partially understood, and most research in the area is aimed at the automated production of compilers. A major problem in the field is deciding in what order to evaluate the attributes. Since symbol tables are treated as attributes, another problem is avoiding having to copy symbol tables (which are large objects) up and down the tree.

Only very recently [Demers 1981, Reps 1981] has there been any attention to the problem of incremental attribute evaluation, which is required in an interactive editing system such as Babel. The Reps paper solves some of the problems of incremental attribute evaluation nicely, and Babel has built upon it. His method was designed with the Cornell Synthesizer in mind, and required adaptation for use in an incremental parsing system.

2.11.1. Problems with Attribute Grammars

In producing any system based on attribute grammars, one must be very careful to have a sound theoretical model. For example, circular grammars[4] should be detected and rejected.

It is also important not to waste any attribute evaluation effort. Any system that reevaluates the same attribute more than once during a single error check is wasteful. Such wasteful reevaluations are quite possible in a careless system design. For example, in figure 2.1, evaluating the attributes in depth-first order could produce the evaluations of a, b, d, e, ..., c, b, d, e, ..., causing double the number of evaluations that are needed.

In an incremental system, it is also important to evaluate only those attributes that must be reevaluated. If a change is made to an executable section of a program, usually only that statement or expression needs to be reevaluated. Yet, which a declaration is changed, the entire scope of that declaration must be checked for new errors or correction of old errors.

In a typical attribute grammar, the attribute passed up and down the tree most often is a symbol table. Declarations build a symbol table, one entry at a time, and pass it up the

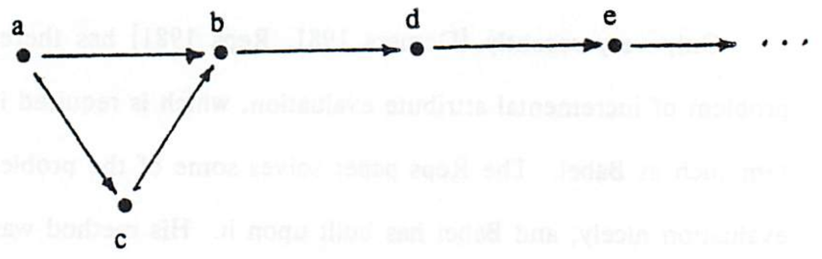

Figure 2.1. Example of Non-Optimal Propagation[5]

[4]A circular attribute grammar is one where an attribute can depend on itself indirectly, thus making an evaluation order impossible to find.

[5]This example is from [Reps1981a]

tree to the root of an executable section of the program. It is then passed down the executable subtree to the leaves, where it is used to look up identifiers. Symbol tables are usually large, and the attribute grammar formalism requires that each attribute be copied at each step in the tree. Such copying clearly makes inefficient use of both time and space.

Some method of sharing symbol tables among attributes is needed. The simplest solution would be to have a static symbol table, and to pass a pointer to it around the tree. Such a solution is, in effect, implemented in most compilers, and is suitable to attribute grammar driven compilers as well.[6]

Unfortunately, this method does not work with incremental systems. To see why, consider the deletion problem, in which the declaration of an identifier is deleted. A system based on a static symbol table will have attribute rules for declarations which *add* the semantics of a declaration to the symbol table. There can be no rule for what to do when a declaration is deleted, since rules are defined only for branches of the tree that exist. There is no way to include a rule on a production to *delete* the symbol table entry, since the production it goes with is no longer a part of the tree. As a result, the entry for the deleted identifier will remain in the symbol table, and uses of it will be incorrectly considered correct (if there were no other declarations of that identifier) or incorrectly scoped (if there was a declaration of the same identifier in an outer scope).

Since symbol tables are such an important part of the semantics of programming languages, it is important to provide a high level symbol table facility in any system handling semantics. Not only does the hand-coding of the symbol table produce a faster system, but a high level interface means that the language implementor can produce a working implementation more quickly, since it is not necessary to code and debug details of the symbol table implementation.

[6]For example, the MUG2 system [Geigerich 1979] allows the implementor to make non-local references to a global symbol table pointer.

However, the scoping rules of languages differ. Some languages are block structured. Others have different symbol tables for different kinds of identifiers. For example, record fields are usually drawn from different pools than ordinary variables. In order to accurately represent such rules, a symbol table tool must not be overly inflexible. It must be possible to conveniently represent the symbol tables of block structured languages such as Algol 60 and Pascal. Babel's *symbol table building block*, described in section 5.8, is a solution to these problems.

2.12. Reps

Reps [Reps 1981] has begun to investigate algorithms for incremental evaluation of attribute grammars, in preparation for a multi-language version of the Cornell Synthesizer. Reps assumes the tree is kept "prepared for propagation at the editing cursor". Preparation consists of keeping either a *characteristic subordinate graph* or *characteristic superior graph* available on each node in the tree, depending on whether the node is below or above the cursor, respectively. These graphs summarize the attribute activity below or above the node to which they are attached. The characteristic subordinate graph of a node n is the graph (V,E) where V is the set of attributes of the node n, and $e=(v1,v2) \in E$ iff v2 depends on v1 *indirectly* by some chain of dependencies below the node n. The characteristic superior graph is similarly defined for dependencies above the node n.

A graph, M, of attributes needing evaluation and their interdependencies is maintained by the algorithm. (M is always a subgraph of the interdependency graph for the entire tree.) The edges of M are of two kinds: *direct* edges representing direct dependencies, and *path* edges representing indirect dependencies.

In Reps' algorithm, all changes are assumed to consist of grafting or pruning a subtree at the editing cursor. When a change is made, M is initialized to be the union of the characteristic subordinate graph and characteristic superior graph at the editing cursor. An attribute to evaluate is chosen from the vertices of M that have in-degree zero.

After evaluation, if the value of the attribute has changed, and if there are any path edges leading from the attribute vertex in M, M is expanded to account for more attributes which must now be reevaluated. The expansion consists of taking the union of the dependency graph D[p] for the appropriate production, and the appropriate characteristic graph of the children or parent node. (The production and graph chosen are either the production above the node or below it, and the superior or subordinate, depending on whether the attribute is inherited or synthesized.) Edges added are direct edges if they came from the dependency graph, or path edges if they came from the characteristic graph.

Since the tree must be kept prepared for propagation at the cursor, when the cursor is moved, it is necessary to recompute some characteristic graphs. For each single cursor motion to a node adjacent in the tree, these computations can be done in constant time.

2.13. Stallman's Letter

Richard Stallman wrote a letter to Computing Surveys [Stallman 1978] arguing that text editors are superior to list structure editors in a Lisp environment. He cites eight advantages of text editors. It is our claim that while a user of Babel is actually editing a tree structure (no text copy is kept) all of the problems listed by Stallman have been solved.

1) The user can specify any style of indentation and the system will never override it.

2) Comments are easily stored and formatted as the user likes them.

3) The user can create unbalanced parentheses while editing a function. ... The user can also move, delete, or copy blocks of syntactically unbalanced text.

4) The editor can provide commands to move over balanced objects or delete them.

5) A text editor can support extended syntax. For example, ... 'FOO is equivalent to (QUOTE FOO).

6) A text editor can be used for languages other than Lisp, including English.

7) With a structure editor, temporary semantic bugs can be dangerous. In editing the system or the editor, one cannot introduce a bug one moment and fix it the next without risking a crash. But in editing text, changes take no effect until the user gives the command.

8) The editing commands most natural for use on a display terminal are those whose meaning is obvious in terms of the displayed text. A data structure of text is natural for them, but implementing them in a structure editor would be very difficult. There are few screen-oriented structure editors.

2.14. Comparison of Babel with Other Systems

This section compares elements of various recent program editors. See Figure 2.2 for a summary. In the figure, "yes" means the feature is provided, "no" means the feature is not addressed, and "n/a" means it is not needed. Other comments are explained in more detail in the text.

2.14.1. Language

Most of the program editors are for a single language. Even though PCM [Yonke 1975] and CAPS [Wilcox 1976] are among the oldest projects, they are truly multi-language editors. Most of the authors of recent single language editors appear to be unaware of PCM. CAPS has virtually escaped attention, in spite of its description in the *Communications of the ACM.* PCM was tested only on Pascal. CAPS was tested on small subsets of Fortran, PL/I, and Cobol. IPE [Feiler 1980] was tested on a dialect of C called GC. Babel has a partial syntax and semantic checking implementation for Pascal, a full syntax and semantic checking for Asple [Cleaveland 1973], and syntax checking implementations of Ada [Ichbiah 1980], LDL (see chapter 6), Rigel [Rowe 1979], Lisp, and a subset of C [Kernighan 1978].

Subject	Interlisp	CAPS	Mentor	PL/CS	LISPEDIT	PCM	IPE	Babel
Author	Teitelman	Wilcox et.al.	Donzeau-Gouge et.al.	Teitelbaum	Alberga et.al.	Yonke	Feiler	Horton
Date	1970	1976	1979	1979	1980	1975	1980	1981
Language	Lisp	Multi (PL/I)	Pascal	PL/I	Lisp	Multi (Pascal)	Multi (C)	Multi
Lexical	Lisp	hand coded tables	hand coded	hand coded	Lisp	Lisp atoms	n/a	Lex reg. expr.
Comments	Lisp S-expr	yes	on nodes	in syntax	footnotes	no	no	textual
Syntax	S-expr	recursive descent	some parser	grammar, no parser	LALR	patterns	grammar, no parser	LALR
Semantics	at run time	some	some	full	full	some	full external	varies
Pretty printing	Lisp rules on the fly	text editor	Pascal Rules on the fly	simple rules from grammar	priority, on the fly	general, on the fly	rules in grammar	rules in grammar
Input	parse goal driven	text editor	parse goal driven	construc-ted, exprs parsed	text, incr parse	parse, goal driven	constructed	text
Incomplete?	no	yes	no	yes	no	no	yes	no
Incorrect?	no	yes	no	no	yes	no	no	yes
Screen	no	no	yes	yes	yes	no	yes	yes
Degree of Use	heavy	light	some	heavy	some	none	new	new
Elision	by level	no	by level	manual	by distance from focus	first & last children	no	off-edge or priority
Preprocessor	no	no	no	no	no	no	no	no
Compiler	yes	no	no	no	yes	no	yes	no
Interpreter	yes	yes	no	yes	yes	no	no	no
Debugger	yes	yes	no	yes	yes	no	yes	no

Figure 2.2 Summary of Recent Program Editors

2.14.2. Lexical Scanning

Few of the systems address the lexical issues that vary so much from language to language. The single language systems all have hand coded scanners, if they have scanners at all. PL/CS and IPE do not need scanners since the user explicitly delimits all input. (PL/CS does scan and parse expressions using conventional methods.) PCM uses the Lisp parser to do the scanning. CAPS uses a state transition matrix based on regular expressions. Babel is driven by tables generated by Lex.

2.14.3. Lexical Comments

This is another area that has been largely ignored by the literature. In practice, being able to place comments in a program is essential for a programming tool to be useful. Some systems simply do not allow comments. Interlisp treats comments as a special form of S-expression, as do most Lisp systems. PL/CS considers comments to be a part of the syntax of PL/I. LISPEDIT views comments as footnotes attached to a spot in the program, marking them with a footnote marker, and displaying them where space permits. Mentor stores comments as attributes of nodes in the abstract tree, flagging only whether they came before or after the node. Babel treats comments as part of the text of the program, attaching them in the tree to the following token.

2.14.4. Syntax and Parsing

The Lisp systems use a standard Lisp parser, except for LISPEDIT which uses an LALR(1) parser. The constructive systems PL/CS and IPE do not need to parse, since the user is building structures. PL/CS does parse expressions to cut down on tedium. Mentor uses an unidentified goal-driven parser. PCM uses a technique that is closer to pattern matching than parsing. CAPS parses with recursive descent procedures, written in a special assembly level language that the system interprets. Some degree of incrementality is present in CAPS due to the preservation of a *state vector* and the possibility of backing up to a known state and starting from there. Babel parses incrementally using an LALR(1) parser.

2.14.5. Semantics

Some of the systems (Interlisp, PL/CS, LISPEDIT) are complete programming environments, and have the semantics hand coded for the particular language, both at compile time and run time. IPE uses hand-coded, externally linked procedures called whenever a user procedure that has been changed is exited. Mentor, PCM, and Babel are editors and attempt varying degrees of semantic analysis. Mentor does not do semantic checking automatically, but Pascal-specific semantic routines to do such things as program transformations and error checks have been written in a tree-oriented specialized language called MENTOL. PCM does limited semantic checking, suitable mainly for catching undeclared variables in block structured languages. Such checking is probably the single most useful check that can be made. CAPS provides simple commands in the parsing language for manipulating the symbol table, and has a powerful CAI error diagnostician to help a programmer find the cause of the error. Babel provides an attribute grammar based language so that the implementor can check as much or as little as desired.

2.14.6. Prettyprinting

Most of the tree editors store only trees and cannot remember how the user originally indented the program. They prettyprint the program, or a section of it, every time the program is redisplayed. Such an approach is very useful in practice, since it prevents users from deluding themselves by mismatching brackets. However, users of text editors are used to their own styles and may have difficulty adjusting to an enforced style. Often, there are problems with prettyprinting comments, and with unusual situations involving very long or short lines. Whether the ability to format one's program as desired is good is a highly emotional subject. See [Sandewall 1978] and [Stallman 1978] for both sides of this issue.

The Babel system takes a new approach by giving the user all the flexibility of text while keeping only a tree representation. It is our claim that all the disadvantages cited in [Stallman 1978] are technically solved by Babel, providing an interface with none of the disadvantages of a tree editor, yet with many of the advantages. Babel includes prettyprinting rules in the language descriptions, which can be used to drive a prettyprinter.

2.14.7. Form of Input

Most of the systems allow the user to textually type in portions of the program that are to be changed, with some restrictions. Mentor requires the user to specify which non-terminal is being entered. CAPS keeps a text representation of the program to which the user appends with ordinary text editor commands. Interlisp, Mentor, and PCM in effect incrementally parse by figuring out where to put the new trees generated. The parsing techniques used are goal driven, however, and so do not allow insertion of arbitrary text, such as "**begin** ; **end**" Babel and LISPEDIT do true, LR incremental parsing, and allow arbitrary changes to the program.

The constructive systems PL/CS and IPE take a completely different approach, requiring the user to build the tree structure one node at a time. Whether such a system is effective remains to be seen. Only PL/CS has been in production use, primarily by introductory programming students who have never used a text editor, and it does parse expressions. In the author's opinion, a well designed constructive system could cut down on the number of keystrokes typed by an experienced programmer, and possibly cut down on input errors, but would be substantially more difficult for a new user to learn. Many of the advantages of a template based system can be added to Babel in the form of language dependent macros, as discussed in chapter 4.

2.14.8. Style used to Change a Program

The commands used to make changes to a program differ dramatically among the different editors. Consider, for example, the program fragment:

```
a := b;
return c;
```

which the user wishes to make conditional:

```
if b < 0 then
begin
        a := b;
        return c;
    end;
```

In Mentor, and PCM, it is necessary to delete the two statements (presumably into a logical register to avoid retyping them). Then the user textually enters the code being added:

```
    if b < 0 then
    begin
    end;
```

Finally, the deleted text is replaced into the program between the **begin** and **end**.

In Interlisp, commands exist that make changes to the tree structure that appear to move, add, and delete parentheses. These commands can be used in the above example to avoid retyping of text.

In PL/CS and IPE, the change is even more tree-specific. The user would delete the two lines into a logical register, as before. Then nodes for **if**, <, **variable**, and **constant** are created manually. (The **begin end** nodes need not be manually entered because the systems always provide such brackets where they are options.) The cursor is then moved from the 0 through the tree to the empty statement sequence between the **begin** and **end**. Finally, the deleted two lines would be put back into the tree.

In Babel, as in any text editor, the change is very simple. The cursor is positioned above the assignment, the **if** and **begin** lines are entered, the cursor is positioned to after the **return**, and the **end** is entered. The user then adjusts the indenting of the two newly embedded statements. CAPS and LISPEDIT share this style of change. In LISPEDIT, it is not necessary (or possible) to adjust the indenting of the two statements, since the system always prettyprints each time it displays.

2.14.9. Incomplete or Incorrect Programs

An incomplete program is one in which in the tree representation, a nonterminal has no expansion (i.e., subtree). Such a nonterminal is a *stub* or *hole* to be filled in later.

An incorrect program is one that does not conform to the grammar, that is, a program that has a syntax error. An incomplete program is not necessarily incorrect, because stubs are viewed as the nonterminal the stub represents. Similarly, an incorrect program may or may not be incomplete.

The constructive systems allow incomplete programs, but do not allow incorrect programs. This is usually cited as one of the advantages of constructive systems, since syntax errors cannot occur. The other systems somehow arrange that incomplete programs never occur.

The Lisp systems always have complete S-expressions with balanced parentheses. PCM asks the user for corrections to input containing syntax errors. CAPS requires that the program be complete and correct to the left of the cursor, but does not check to the right. LISPEDIT treats unparseable input as a comment until the next parse.

Babel does not allow programs which are incomplete in this syntactic sense. If the user enters a program he considers incomplete, and the program contains syntax errors because of the missing text, the editor will consider the program to be incorrect. All incorrect programs are handled in a uniform way. The erroneous portion of the program is highlighted on the terminal screen, and the incorrect program is accepted for further processing. Note that it is not possible in Babel for the programmer to place unexpanded nonterminals in the tree. Thus it is never possible to create an incomplete program as defined at the beginning of this section.

2.14.10. Screen Orientation

The last decade has seen a tremendous change in computing hardware. Printing terminals are rapidly being replaced by high speed CRT screen terminals. Editors are becoming screen-oriented, too, since showing the user what is really there helps prevent misconceptions from being formed and saves the user from having to print out context frequently.

Most recent editors are screen-oriented, both text and tree editors included. CAPS, PL/CS, LISPEDIT, IPE, and Babel are all screen-oriented. (Sandewall and Stallman seem to feel that tree editors are more difficult to make screen-oriented than text editors. The state of the art has since progressed to the point where this is no longer true.)

2.14.11. Elision

When the entire program does not fit on the screen, the editor must decide what to leave out. Text editors have only the choice of not displaying what is off the edge of the screen. In principle, tree editors can do a much better job of elision. Most of the editors here do some kind of elision.

Mentor recursively prints a subtree to whatever depth is requested by the user. This technique is suitable largely because the size of the screen is not taken into account, since Mentor is not screen-oriented. Interlisp uses an elision technique similar to that of Mentor. PCM elides a subtree by printing only the first and last tokens of the subtree. PL/CS allows the user to manually elide a subtree, and otherwise uses off-the-edge elision. LISPEDIT uses a complex priority algorithm every time the screen is redrawn, which happens every time a line is typed or the focus is moved. Interlisp does not elide at all but, typically, in a Lisp environment functions are small enough so that this doesn't matter much. Babel gives the user a choice of off-the-edge elision or priority elision.

2.14.12. Preprocessor

Several current languages (C, Bliss, PL/I) have a preprocessor that is applied before the scanner. In all but C, the preprocessor is non-essential. However, the programming style of C is such that all but the most trivial C programs use the C preprocessor. Hence, it is impractical to ignore the preprocessor in a system such as Babel in the UNIX environment. No other system has addressed the issue of preprocessors. Babel's description of C has some very simple provisions for handling programs making simple use of the C preprocessor.

2.14.13. Degree of Use

Only Interlisp and PL/CS from this group have been used heavily. Interlisp has been used since 1970 for production Lisp work on Tenex systems, and was the implementation language for PCM. PL/CS is used for introductory programming courses at Cornell and elsewhere. CAPS was used for an introductory CAI programming course at Illinois. LISPEDIT is used internally by a dozen or so persons at IBM. Mentor has been distributed to a number of institutions and is used by the authors. PCM was an experimental project that has not been seriously used. IPE and Babel are still too new to have been used by many people.

2.14.14. Execution

Some of the systems in this group are complete programming environments. Such systems provide compilers, interpreters, and/or debuggers. The Interlisp, CAPS, PL/CS, LISPEDIT, and IPE systems include interpreters which run the user's program without leaving the system. Mentor and PCM, do not provide runtime facilities. Babel does not provide such features, since it is primarily an editor. The system could be extended to include these features. The amount of effort would be typical of the back end of a conventional translator implementation.

Babel is intended as a set of tools to support an intelligent editor. The user is expected to exit the editor (after being assured by the editor that his program contains no static errors) and to run a compiler, translating from the tree file produced by the editor into machine code. A number of compiler schemes are possible. One possibility is to unparse the tree, producing a conventional text file, and to run a conventional compiler on this text. Another option is for a compiler to work directly from the tree file. This option eliminates the need for the compiler to scan, parse, and generate a symbol table, and makes error detection and recovery unnecessary. A third option would be for an incremental compiler to recompile only what has been changed. The tree structure makes it easy for an incremental compiler to determine what has been changed. A fourth possibility is to carry the incremental semantic evaluation to the point of producing intermediate code in the tree, and interpreting this code directly, or to generate code from this intermediate code.

CHAPTER 3

Notation

This chapter contains definitions of technical terms used in this dissertation. All definitions are collected here for ease of reference.

3.1. Languages

A **string** is a sequence of **symbols** from an **alphabet**.

A **language** is a set of strings that are acceptable to a particular computer program. This concept includes not only languages traditionally considered **programming languages**, but the languages accepted by other programs including document preparation languages such as Troff [Ossanna 1976] or Scribe [Reid 1980], computer assisted instruction languages such as Learn [Kernighan 1979], program maintenance languages such as that understood by Make [Feldman 1978], and miscellaneous data files which are stored as text. Babel can be used for both programming and non programming languages, provided the languages meet the restrictions set forth subsequently.

3.2. Grammars

A **context free grammar (grammar)** is a four-tuple (V, N, S, P) where V is the finite vocabulary of **nonterminal symbols (nonterminals)** and **terminal symbols (tokens)**[1], $N \subseteq V$ is the set of **nonterminals**, $S \in N$ is the start symbol, and P is the set of production rules of the form

$$lhs : rhs$$

where $lhs \in N$, $rhs \in V^*$. *Lhs* is the **left hand side** of the production, *rhs* is the **right hand**

[1]Terminal symbols are called **tokens** rather than **terminals** to avoid confusion with **computer terminals**, which are input/output devices for computers. In this dissertation, **terminal** always refers to the input/output device.

side. V^* is the transitive closure of V. A **symbol** is any $v \in V$. An **empty production** is a production with no symbols on the right hand side.

3.3. Attribute Grammars

An **attribute grammar** is a five-tuple (V, N, S, A, P) where V, N, and S are as in a context free grammar; for each symbol v in V, A(v) is the set of attributes of v; P is a set of productions, p, of the form

$$lhs : rhs \ rules$$

where *lhs* and *rhs* are as in a context free grammar, and the rules are of the form

$$a_{s_0 i_0} \leftarrow f_{psi}(a_{s_1 i_1} \cdots a_{s_{n_p} i_{n_p}})$$

where there are n_p symbols in *rhs*, and a_{si} is the i^{th} attribute of symbol $s, 0 \leqslant s \leqslant n_p$, of production p. $s=0$ refers to *lhs*, $s>0$ refers to the s^{th} symbol of *rhs*.

Each attribute $a \in A(v)$, for each $v \in V$, is classified as either **inherited** or **synthesized**. For each symbol s of production p, the set of attributes of s given a value in that production must be the synthesized attributes (if $s=0$), or the inherited attributes (if $s>0$). The restriction on the nature of the attributes is made not for any implementation reason, but because attribute grammars that do not meet the restriction do not make sense.

3.4. Program Trees

A **node** is a data object having associated with it a small positive integer called the **type**, and some **data** which depends on the type. Types include integers representing tokens, nonterminals, attributes, comments, scanner errors, etc. A **nonterminal node** is a node whose type is nonterminal, a **token node** is a node whose type is token, and so on. Where the meaning is clear, such nodes will be referred to as **nonterminals**, **tokens**, and so on. The size of the data depends on the type and data, and is fixed at the time of creation of the node. (For example, the data in a token node includes the text of the token, whose length is fixed only by the kind of token.) A **tree pointer** is a reference to a node.

A **program tree** (**tree**) is a collection of nodes, including one particular node called the **root**. The other nodes are partitioned into a sequence of disjoint groups $t_1, \ldots, t_n, n \geqslant 0$, each of which is a tree. The t_i are **children** of the root. The root is the **parent** of the t_i; t_1 is the **first child** of the root, and t_n is the **last child**. For each $i, 1 \leqslant i < n, t_i$ is the **right sibling** of t_{i-1}, and t_{i-1} is the **left sibling** of t_i.

There is one particular program tree associated with the system at any time. This tree is usually referred to as **the tree**. It contains a representation of the program being edited. One portion of the tree is a **parse tree** made up of nonterminal nodes and token nodes, corresponding to the grammar for the language in use. This tree is stored on disk, and portions are brought in to primary memory when needed.

Nodes have associated with them five **neighbors**; the **parent**, **first child**, **last child**, **right sibling**, and **left sibling**. Tree pointers to these five neighbors are stored in each node, if they are defined by the above definitions. Otherwise, the special value **tnull** is stored, referencing "no node".

A **path** is a sequence of nodes, $t_1, \ldots, t_n$, $n>0$, such that t_i and $t_{i+1}, 1 \leqslant i < n$, are neighbors. A **firewall** is a sequence of nodes (not a path) $l_1, \ldots, l_a$, A, $m_1, \ldots, m_b$, Z, $r_1, \ldots, r_c$, where l_1 is the first child of the root, r_c is the last child of the root, (l_1 and r_c are **endmarkers**) for $1 \leqslant i \leqslant a$, $1 \leqslant j \leqslant c$, l_i and r_j are either nonterminals or tokens, and for $1 \leqslant i \leqslant b$, A, m_i, and Z are tokens. $l_1, \ldots, l_a$ is the **left firewall**, A, $m_1, \ldots, m_b$, Z is the **middle firewall**, and $r_1, \ldots, r_c$ is the **right firewall**.[2]

The firewall divides the tree into three parts: parts on, above, and below the firewall. Formally, a node, n, not on the firewall that is on a path from another node, f, on the firewall, to the root, is **above** the firewall. A node on neither the firewall nor such a path is **below** the firewall. (See figure 3.1 for an illustration.)

[2]For those familiar with parsing terminology, the left and right firewalls always represent the frontier, in a parse from the left or right ends of the program, at the point where A or Z is reached, respectively.

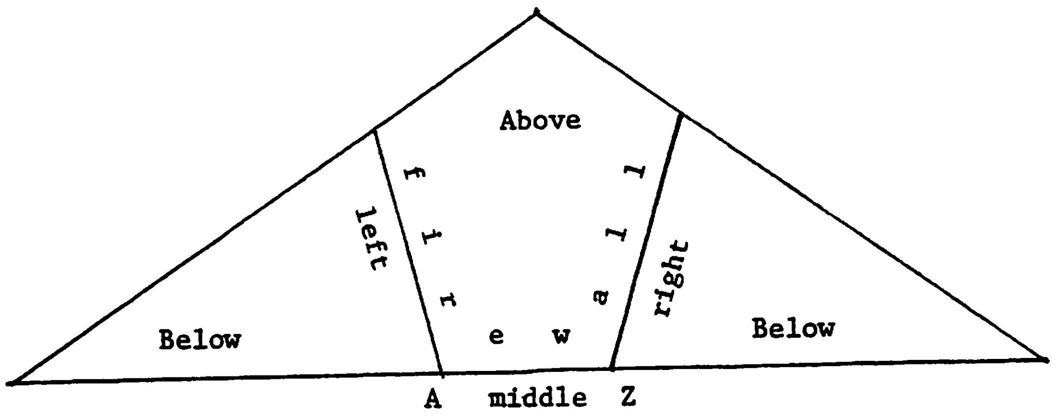

Figure 3.1 Firewall

3.5. Representation of Text

A **newline** is the separation between two lines of a text file. Babel views newlines as characters.

A **utoken** (user level token) is either a token, a comment, or a lexical error. (Lexical errors are treated as comments by Babel.)

Token nodes have associated with them, as part of the data, the particular token number, the text of the token (as a character string), and two integers called the **white space count** and the **newline count**. The newline count is the number of newlines between the token and the textually preceding utoken. The white space count is the number of blank spaces separating the token from the preceding utoken (if the newline count is zero) or the closest preceding newline (if the newline count is positive). The effect of special characters such as tab and backspace upon the position of the token is figured into the white space count. Comments and scanner errors are attached as children of the token they precede, and have their own newline and white space counts.

A **text position** is a tree reference corresponding to a particular character of a text file. It consists of a pair (t,c), where t is a tree pointer to a utoken, and c is an integer. If $c \geqslant 0$, the text position refers to character number c of the text of the utoken (numbering the characters from zero). If $-100 \leqslant c < 0$ the reference is to the $-c^{th}$ blank before the text of the utoken. If $c < -100$ the reference is to the $(-100-i)^{th}$ newline preceding the text of the utoken.

The **fringe** of the tree is the sequence of utokens in the tree, in the order of the textual representation. (This is the order obtained by traversing the tree depth first, visiting a node after its children.)

A program tree has associated with it a text position called the **focus** which represents the current point of interest in the editor. This focus is recorded in the disk file and remains in force after leaving the editor. The focus can also take on nonterminal values $(t,0)$ where t is the nonterminal being referred to. Textually, a nonterminal focus refers to the first character of the leftmost utoken below the focus in the tree. If, due to empty productions, there are no utokens below the focus, the text representation is to the first utoken in the fringe after the focus. (Since endmarkers are always present in the grammar, there is always a next token for a textual representation of the focus, and for attaching comments.)

CHAPTER 4

User Interface

This chapter gives an example that demonstrates some of the capabilities of the Babel Editor. The design of the command language is discussed.

The key observation about the user interface is that the user types and sees what he would when using a typical screen-oriented editor. While the specific command set chosen here is different from existing screen editors such as EMACS [Ciccarelli 1978] and vi [Joy 1980], it would be possible to rewrite the interface to make it look just like one of these, or any other screen-oriented text editor. The user sees a text editor, except that errors that a text editor would be unlikely to detect are pointed out by highlighting the offending part of the program.

4.1. Example Session

In this section, we show an example session with the Babel editor. The user enters a small Pascal program, and then makes modifications to it. Errors made along the way are pointed out. Finally, the program is saved on the disk and the editor is exited.

The true utility of a system such as Babel is with programs that are too large to be understood at a glance. To keep the presentation to a reasonable size, the example must be kept small. Of course, an example this small is easily programmed using any editor. The reader is asked to extrapolate the example to larger problems.

In the display (see Figures 4.1−4.8) the first line is the *echo line* containing the command typed by the user. The second line is the *message line* where system messages from Babel to the user are displayed. Remaining lines are the program itself. Error messages and portions of the program in error are highlighted on the screen. This is indicated here by **large bold face**. The focus (cursor) is indicated by _.

The ~ character is an endmarker. It is shown to the user to make the beginning and end of the buffer more visible[1]. The endmarker should not be confused with the notion of a *fence*, which is a line drawn on the screen separating windows. Even though the echo and command lines can be considered small windows, the current implementation of Babel does not draw fences because lines on a typical 24 line terminal screen are too valuable to waste on fences. While a multi-window system would need fences to avoid confusing the user, the current single window system does not need them. Such a multi-window extension could be added to Babel in a straightforward way.

The user enters the editor, creating a new file with the command

```
be −lpascal copy.t
```

Babel finds no file of that name, so it creates an empty file. The user sees the display of Figure 4.1.

```
~
copy.t (new file): pascal
~
_
```

Figure 4.1 Display upon entry to the editor.

At this point, the user can begin typing in the program. Program lines can be typed in directly. They will be echoed in the echo line as they are being typed. As each line is entered, it is inserted into the file after the line containing the focus. The program is incrementally scanned and parsed and (if syntactically correct and complete) semantically checked after each textual line.

[1]The idea of displaying ~ at the ends of the buffer is from **vi** [Joy 1980].

```
program copy(input, output);
go on...

program copy(input, output);
```

Figure 4.2 Display after entry of one line.

In the example, the user is entering a program to copy the input file to the output file. After the first line of input (see Figure 4.2) the program will not be complete, (i.e., a syntax error was detected at the endmarker), and the user will be warned of this with the "go on" message. The endmarker is highlighted, indicating the location of the syntax error.

The user types in the remainder of the program without event. After the final line has been entered, the parser will accept it. Since the syntax is accepted, the program is semantically checked. In this case, no semantic errors are detected. The user sees the display of Figure 4.3.

```
end.

program copy(input, output);
var
        ch: char;
begin
        while not eof do
        begin
                while not eoln do
                begin
                        read(ch);
                        write(ch);
                end;
                readln;
                writeln;
        end
end.
```

Figure 4.3 Display after entry of the program.

This illustrates one style of program entry: straight top-to-bottom typing in of the program. This is suitable for a situation where the program is already written on a piece of paper. Such a program entry style is impossible in a template based system.

A second style would be more appropriate for program development at the terminal. In this style, the user types in the framework of the program (a very short, correct program) and then adds to it one piece at a time. Since most of the intermediate states will be syntactically and semantically correct, the user gets a good deal of feedback while developing his program. While this program development style is superficially possible with template systems, our system makes it possible to go through syntactically illegal states, and allows complete lines of text (or portions of lines, in the case of editing commands such as **change**) to be typed in exactly as displayed. The remainder of the example presented here illustrates this program development style.

Now that the program has been entered, the user wishes to modify it to print line numbers. Using textual arrow keys, he positions the focus to the newer **begin**, and types in a write statement, producing the program shown in Figure 4.4. This statement uses the variable *line*, which has not been declared, and the editor indicates this by highlighting the variable.

The user, unsure why the editor flagged the variable, positions the cursor to the offending variable and types the "why" command. The editor responds as shown in Figure 4.4.

Continuing to add the line numbering code, the user moves the focus to the first **begin** and types in an assignment statement to initialize the variable *line*. He accidently leaves out the semicolon ending the statement. The system informs him of the syntax error with a message and by highlighting the token where the error was detected (see Figure 4.5). Semantics are not checked when there is a syntax error, so the second instance of the undeclared variable *line* is not detected yet.

```
.why
Undeclared variable

program copy(input, output);
var
        ch: char;
begin
        while not eof do
        begin
                write(line:5, '    ');
                while not eoln do
                begin
                        read(ch);
                        write(ch);
                end;
                readln;
                writeln;
        end
end.
```

Figure 4.4 The users asks for a detailed error message.

```
        line := 1
syntax error

program copy(input, output);
var
        ch: char;
begin
        line := 1
        While not eof do
        begin
                write(line:5, '    ');
                while not eoln do
                begin
                        read(ch);
                        write(ch);
                end;
                readln;
                writeln;
        end
end.
```

Figure 4.5 A syntax error.

This error is corrected with the "change word" command, changing the "1" on the line to "1;". The editor reparses, finding the syntax correct, and checks the semantics. It discovers the second instance of the undeclared variable *line*, which it highlights (Figure 4.6).

```
.cw '1;'

-
program copy(input, output);
var
        ch: char;
begin
        line := 1;
        while not eof do
        begin
                write(line:5, '    ');
                while not eoln do
                begin
                        read(ch);
                        write(ch);
                end;
                readln;
                writeln;
        end
end.
-
```

Figure 4.6 Correction of syntax error.

The user now enters the declaration of the variable *line*, and the syntax and semantics are accepted by the editor (Figure 4.7). Note that the program still contains a logic error (the *line* variable is never incremented), but this error cannot be detected by Babel.

Finally, the user enters the statement to increment *line*, and types the "exit" command to leave the editor (Figure 4.8). The program is written out to disk and control is returned to the top level command interpreter.

At this point, a tree file exists on the disk containing the Pascal program. The program has been scanned and parsed, and a symbol table has been created in the tree file. Since this analysis was done incrementally, there was no single long delay while the user

```
            line: integer;

-
program copy(input, output);
var
            ch: char;
            line: integer;
begin
            line := 1;
            while not eof do
            begin
                        write(line:5, '    ');
                        while not eoln do
                        begin
                                    read(ch);
                                    write(ch);
                        end;
                        readln;
                        writeln;
            end
end.
-
```

Figure 4.7 Correction of a semantic error.

```
.exit

-
program copy(input, output);
var
            ch: char;
            line: integer;
begin
            line := 1;
            while not eof do
            begin
                        write(line:5, '    ');
                        while not eoln do
                        begin
                                    read(ch);
                                    write(ch);
                        end;
                        readln;
                        writeln;
                        line := line + 1;
            end
end.
-
```

Figure 4.8 Leaving the editor.

waited for the program to be processed.

A compiler could start from this tree form, eliminating considerable processing time (since the job of the compiler's front end is already done) and complexity (issues such as parsing method, error recovery, and one pass problems disappear).

Since no such compiler currently exists in our experimental implementation, it is necessary to use an ordinary text compiler instead. The user must convert the tree file to text with

```
bcat copy.t > copy.p
```

and can then use a text compiler. Obviously, a tree editing system such as Babel is at its best in an environment with tree compilers, and creation of such compilers for Babel should be undertaken. Tree compilers and incremental compilers already exist, such as that described in [Feiler 1980], and are a valuable addition to any programming environment.

4.2. Design of the Command Language

The command language is not closely tied to the remainder of the editor. Any text-oriented editor interface could be implemented on the Babel system. The particular command language chosen here was designed with two goals in mind. First, the language should be easy for a beginner to learn. Second, the interface should be easy to implement.

A beginning user on a computer system must usually learn several things at once. He must learn the top level command language, the command language for an editor, and the language accepted by the program for which it is being prepared (i.e., a compiler or text formatter). In order to make this task simpler, the Babel editor command language has been designed to be compatible with the top level command language of the Unix system on which it runs, the *shell* While out of necessity, the individual commands differ, the syntax for commands is the same in the editor and the shell. Commands are sequences of words, separated by blanks or tabs. The first word is the command, the remaining words are arguments to the command. Conventions for specifying options, special characters in arguments, and multiple commands on one line are the same in the editor and in the shell.

Another factor of many text editors which is confusing is the notion of an *input mode*. Having an input mode, as in **vi**, allows the user to edit with fewer keystrokes, since all characters can be used for both commands and text, but confuses most naive users. While Babel has an input mode, the command language is designed in such a way that the user need not ever leave input mode. Thus, a beginner need not worry about input mode, but an experienced user can switch modes if desired.

Established text editors like **EMACS** and **vi** have had years of effort put into finely tuning their command languages, and the resources to duplicate this effort were not available for this project. The option of taking the existing code for **vi** and gluing it to the Babel back end was considered and rejected because the existing code was not designed with such gluing in mind. A rewrite of the user interface would be possible with somewhat less work than was originally put into the existing editors, and is planned for a future version.

When the editor is idle and waiting for input, the echo line contains the prompt '*'. It is not necessary for the user to wait for the prompt, since typeahead is not only understood, but echoed as it is typed in.

A line of text can be typed in directly, causing it to be inserted after the line containing the focus. This is a special case of the **add** command, which appends its argument as a line after the current line. Thus the set of keystrokes "**CMD a d d SPACE ´ b e g i n ´ CR**" can be abbreviated by the special case "**b e g i n CR**". The special case was included only for convenience and to make the editor easy to learn, since the regular **add** command has the same power. In practice, the special case is always used; the only need for the **add** command itself is from inside macros.

In order to distinguish commands from text, the user presses the *command* key before typing the command. A period is echoed to indicate that a command is being typed in.[2]

Since the command key must be pressed before commands but not before lines of text, the user is in input mode. This mode is the default, and it is quite possible to use the editor extensively without changing modes. This property makes the Babel editor easy for a beginner to learn.

A user with many changes to make might prefer, by default, to have his input line treated as a command, and to type a special key before text instead. It is possible to enter *command mode* to arrange for this to be the case. In command mode, the period is still echoed for commands and omitted for text. Not only does this make it easy for a user to determine which mode the editor is in, but it also provides a consistent display format.

Always showing text with no leading period assures that text typed into the echo line will line up properly with other text on the screen, avoiding the common "off by one" problem caused by the width of the prompt or insert command in many systems. The * prompt of Babel is erased when the user begins to type in a command, avoiding another common problem: the "ghost prompt" problem caused by typeahead. This problem occurs when the user types in two commands, one command completes and a prompt is printed. Seemingly the system is ready to accept input, but it is really processing the second command. Almost any good screen-oriented editor will avoid these problems by not echoing typeahead until it is processed. Babel avoids the problem and still lets the user see his typeahead as it is being typed.

Commands are built in to the editor to move the focus around in the tree. "Textual arrow key" commands **up**, **down**, **left**, and **right** move the focus one character in the indicated direction on the terminal screen, exactly as in a text editor. "Tree arrow key" commands **in**, **out**, **next**, and **previous** move the focus in the tree to the first child, parent, left sibling, and right sibling, respectively, of the old focus. If the focus is on the fringe of the

[2]The command key is initially BREAK. It can be set by the user to any control key.

tree, it can be on any character position of a token. If the focus is on a nonterminal, only one position on the nonterminal is possible, since character positions on the nonterminal are not defined. In this case, the cursor is displayed at the beginning of the leftmost token below the focus.

It would be tedious to have to type the sequence "**CMD r i g h t CR**" repeatedly to move the focus to the right several spaces. To make movement of the focus convenient, a *macro* mechanism is provided in **be**. This maps single keystroke commands into full editor commands. For example, the keystroke "control R" is mapped into the command "right"; thus to move right several spaces the user need only press control R repeatedly. Similarly, there are control characters for the other seven "arrow keys" described above, as well as other common editor commands.

In addition, if the terminal has true arrow keys (keys labeled with arrows in the four textual directions which transmit recognizable codes), these keys are mapped into the left, right, up, and down commands. Thus, the cursor can be positioned with arrow keys exactly as in **vi**.

In addition to the predefined macros, the user can define other macros, or redefine existing macro keys. This permits the customization of the editor to individual tastes, with almost no extra effort beyond that needed to implement arrow keys.

A natural extension to this concept, which has not been implemented, is the notion of *language dependent macros*. Such macros would be defined when the tables for a language are read. This would provide similar functionality to the template building commands of [Teitelbaum 1979] and [Feiler 1980]. The convention could be adopted that a language always defined macros called **if**, **if-else**, (or a more compact abbreviation), **loop**, **while**, **for**, **proc**, **func**, **decl**, and so on. Since macros expand to an arbitrary command line, a typical macro definition for an Ada **if** might be

add "if expression then" ; add "end if"

expanding to a short, syntactically correct construction. Not only could this save keystrokes for users experienced in a language, but together with standardization of the names across many languages, it could help a user unfamiliar with the syntax of a language get the construct desired. All of this is optional, however, and a user preferring to type in the program text directly could still do so.

Note that the "expression" in the above macro definition would not represent an unexpanded nonterminal, as it would in a template editor, but rather an undeclared variable called "expression". Since the variable is likely to be highlighted (because it isn't declared) the user is still reminded that it must be expanded. For the purposes of implementation of language dependent macros, it might be slightly cleaner to allow unexpanded nonterminals, but it was felt that the added complexity of the user interface was not justified. It would be necessary to be able to textually enter and change these nonterminals, requiring a text representation and rules to disambiguate, in a language independent manner, between tokens and nonterminals. Positioning the focus to the expression and typing a textual "change word" command is just as convenient for the user as in a template editor. The language dependent macro could even include commands to move the focus to the first letter of "expression".

CHAPTER 5

Implementation

5.1. Introduction

This chapter discusses the implementation of the Babel editor. The parts of the system are outlined, and the algorithms used are described.

The basic parts of the system are the *control loop* which controls the rest of the editor; the *keyboard handler* which accepts input from the keyboard; the *command parser* which accepts input from the keyboard handler and determines which command to call with what arguments; the *display processor* which decides what character representation to display in the editor window; the *screen handler* which deals with the CRT screen; the *command routines* which each implement one user level command; the *tree editor* which presents a text interface to command routines; the *incremental scanner* which breaks up text into a list of tokens; the *incremental parser* which restructures the tree according to the grammar; the *incremental semantic evaluator* which applies semantic checks to the user program; the *symbol table module* which implements the notion of a *symbol table building block*, and the *Lisp interpreter* to interpret the attribute evaluation functions. Of these parts, the keyboard handler, command parser, screen handler, and Lisp interpreter are quite ordinary and are not described here. The symbol table interface is described in chapter 6.

5.2. Overall Control

The overall structure of the editor can be viewed as two processes of differing priority. The higher priority process reads and echoes commands from the keyboard. The lower priority process interprets the commands, modifies the tree, and updates the display.

In reality, there is only one process. (Dealing with asynchronous interrupts causes many unnecessary problems with critical sections and shared data structures.) There is a queue of commands to be executed. Whenever the processor is available, be takes one item from the queue, and processes it. An entire tree-changing command can take significant real time, but it is made up of small operations (parsing steps, attribute evaluations, display node checks). After each small operation, the system checks for typeahead,[1] and processes it, so the user gets good response on the keyboard even though there may be a large amount of semantic processing to do.

5.3. Tree Structure

The Babel tree is a disk file consisting of a set of nodes, linked together to form a tree. Each node has a *type*, such as *token*, *nonterminal*, and *attribute*. This representation has all the power of an ordinary parse tree, and is a generalization of that concept. The portion consisting of the *nonterminal* and *token* nodes is the LR parse tree of the program being represented. In addition, other programs needing to place additional information in the tree can attach nodes of different types anywhere in the tree.[2]

There are a number of other existing tree representations used by language implementors. Many compilers build an abstract syntax tree in the process of compiling. Other tree editing systems have internal tree forms. Intermediate forms for the Ada language [Ichbiah 1980] such as Diana [Goos 1981] and its precursors TCOL_{Ada} [Brosgol 1980] and AIDA [Persch 1980] are being standardized. In order to convert an existing compiler to use Babel trees for input (and save the repeated work of scanning, parsing, building a symbol table, etc) it would be easier if the tree representation were the same.

[1]Checking for typeahead is not possible on many systems, such as standard UNIX, but is often available as a local modification. An efficient check for typeahead makes be considerably more responsive.

[2]The convention that a program will ignore any nodes it does not recognize is adopted. This permits new node kinds to be created without requiring changes to existing software.

Such existing tree representations were rejected for two reasons. First, the tree structures existing elsewhere are all based on various abstract trees, not the LR parse tree. There is currently no general purpose algorithm for incremental parsing from an abstract tree. Second, the other representations have no provision for insertion of extra nodes. For example, comments are extra nodes attached to the token they precede. They are very simple to handle in the Babel tree, while the standard representations have difficulty handling them.

In Diana, for example, there is no provision for recreating the source program exactly. While the authors had this in mind when they specified the **source-position** attribute as a standard attribute on most tree nodes, their aim appears to have been primarily to produce error messages referring to the correct location in the source program. They require source-positions on nonterminals, without specifying the meaning intended, and yet do not provide for recording the source positions of tokens which are not explicitly represented in the tree, such as **if**. Comments are attached to a node either before it or after it, making it impossible to exactly reconstruct an arbitrary comment. Such a representation might be suitable for another language editor that did not exactly represent the text typed in by the user, but it cannot be used for Babel.

The Babel system pays a high price for use of the LR parse tree. If an abstract tree mechanism could be used, the resulting trees would be considerably smaller, resulting in smaller disk files and a faster editor. If a method were to be found for general purpose incremental parsing of abstract trees, the Babel system could be modified easily to use it. For such an abstract tree to be suitable for Babel, it would have to represent *all* tokens as leaves of the tree, not just those tokens carrying semantic information. It would also be essential to allow extra information to be attached to the tree, without being recognized by the other parts of the system, as is true in the current representation.

General incremental parsing of abstract trees is potentially a difficult problem. In the process of compression of the tree, much information is lost. Punctuating tokens, such as if and parenthesis, are usually removed. Chain reductions are removed. The tree is usually restructured into a form convenient for the compiler writer, not the parser. Some of the existing systems [Yonke 1975, Donzeau-Gouge 1980] parse using abstract trees, but these parsers are goal driven, and never attempt to merge newly typed text with existing text, restructuring the results as required by the grammar. While the smaller, more natural abstract trees would be preferable to the LR trees currently used, additional research to find a general method for incremental parsing of abstract trees is needed.

Each tree node contains its type, pointers to five neighboring nodes (its parent, left sibling, right sibling, first child, and last child), and type dependent information (such as the text of a token, or the nonterminal number for a nonterminal).

An important property of the implementation is that while there are primitive routines to *retrieve* the values of the neighboring pointers, there are no primitives to *change* them. All structure changes must go through two routines *insert(n, p, l)* which inserts a given node *n* as a child of *p* and a left sibling of *l*, and *prune(n)* which deletes the node *n* from the tree. This requirement has three advantages. First, such a high level interface insures that the tree structure will be consistent at all times, eliminating a large class of editor bugs. Second, these routines, which are inverses, keep a history list of all such changes, so that the *undo* and *redo* commands can be implemented easily. Third, changes to the representation are possible with no change to most of the code.

5.4. Display Algorithms

Display algorithms are given the size of an area on the screen, and read access to the tree and focus. They produce a character representation of some portion of the tree, near the focus, that fits in an area of the given size. Three such algorithms are implemented in the Babel editor.

5.4.1. Recursive Display Algorithm

The recursive display algorithm recursively visits every node in the tree, displaying token nodes as they are found. The algorithm has the advantages of simplicity and speed. It also has a serious problem: what to do when the entire tree does not fit in the given area. Currently, after running off the end of the screen, the algorithm stops. This causes only the first screenful of the program to be visible.

Such an algorithm is clearly unsuitable for an editor. It is, however, well suited to an unparser, and is used for the **bcat** utility which prints textual representations of Babel trees.

5.4.2. Off-Edge Display Algorithm

The off-edge display algorithm prints all of the tree that is within a half screenful of the focus. It effectively elides all of the tree which is "off the edge" of the screen. Most screen-oriented text editors use an approach similar to this one.

The implementation of this algorithm is also straightforward. Starting at the focus, the editor moves backward along the fringe of the tree, counting newlines, until the count exceeds half the screen size. From this point, it moves forward along the fringe of the tree, printing the tokens that are encountered, until the screen is full.

5.4.3. Priority Elision Algorithm

The priority elision algorithm is by far the most complex and least efficient[3] of the three. It is based loosely on the algorithm in [Alberga 1979]. It shows the user those parts of the tree that are near the focus (in the tree, rather than on the screen) in preference to parts of the tree further from the focus. Sequences of tokens that are not displayed are *elided*, i.e., displayed as "..". The string ".." was chosen for compactness and ease of understanding. Another string, such as "..." or "&" could easily be substituted.

[3]Priority elision requires 1.5 to 2 times as much CPU time as the off-edge display algorithm. It also tends to touch more pages, since it examines more of the tree. Thus, the start-up cost is higher because more of the tree must be paged in. Off-edge elision required 50 lines of C code to implement. Priority elision required 750 lines of C code.

As an example of how the user sees priority elision, consider the C program shown in Figure 5.1. Two possible results of the priority elision algorithm are shown in Figures 5.2 and 5.3. In Figure 5.2, the focus is at the root of the tree (and is displayed at the top end-marker). The text displayed provides a very global view of the program. In Figure 5.3, the focus is on an **if** statement well down in the tree. In this case, the user sees text that is more local to the focus.

Distance between a node and the focus is defined inductively in terms of the path from the focus to the node. Let $t_0, \ldots, t_n$ be the path, where t_0 is the focus and t_n is the node.

The distance to the focus, $D(t_0, t_0)$ is 1.

If t_i and t_{i+1} are adjacent siblings, and $D(t_0, t_i)$ is d, $D(t_0, t_{i+1})$ is $d+1$.

If t_{i+1} is the parent of t_i and $D(t_0, t_i)$ is d, $D(t_0, t_{i+1})$ is $3d$.

If t_i is the parent of t_{i+1} and $D(t_0, t_i)$ is d, $D(t_0, t_{i+1})$ is $5d$.

The expressions $d+1$, $3d$, and $5d$ are those given in [Alberga 1979]. Alberga, et. al. do not explain the significance of these expressions. The intent is to make nodes further away in the tree be considerably lower in priority than nearby nodes. These expressions also favor movement upward in the tree over movement downward. It may not matter what expressions are chosen, as long as they increase the value of d.

The presence of the "sibling" case in the above expressions introduces an ambiguity in the calculation of distance, since more than one path is possible if siblings are considered adjacent. This ambiguity is resolved by the algorithm given below, in effect always using the shortest path. The sibling case can only occur when the path taken from the focus goes upward in the tree, then branches to the side, and finally might go down a different branch.

The algorithm uses two primary data structures. The first is a representation of the *display*, consisting of a doubly linked linear list of elisions and tokens, that will be displayed

```
static char *sccsid = "@(#)rmail.c      4.1 (Berkeley) 10/1/80";
char *index();

main(argc, argv)
char **argv;
{
        char lbuf[512];      /* one line of the message */
        char from[512];      /* accumulated path of sender */
        char ufrom[64];      /* user on remote system */
        char sys[64];        /* a system in path */
        char junk[512];      /* scratchpad */
        char cmd[512];
        char *to, *cp;

        to = argv[1];
        if (argc != 2) {
                fprintf(stderr, "Usage: rmail user\n");
                exit(1);
        }

        for (;;) {
                fgets(lbuf, sizeof lbuf, stdin);
                if (strncmp(lbuf, "From ", 5) && strncmp(lbuf, ">From ", 6))
                        break;
                sscanf(lbuf, "%s %s", junk, ufrom);
                cp = lbuf;
                for (;;) {
                        cp = index(cp+1, 'r');
                        if (cp == NULL)
                                cp = "remote from somewhere";
                        if (strncmp(cp, "remote from ", 12)==0)
                                break;
                }
                sscanf(cp, "remote from %s", sys);
                strcat(from, sys);
                strcat(from, "!");
        }
        strcat(from, ufrom);

        sprintf(cmd, "%s -r%s %s", MAILER, from, to);
        out = popen(cmd, "w");
        fputs(lbuf, out);
        while (fgets(lbuf, sizeof lbuf, stdin))
                fputs(lbuf, out);
        pclose(out);
}

/*
 * Return the ptr in sp at which the character c appears;
 * NULL if not found
 */

char *
index(sp, c)
register char *sp, c;
{
        do {
                if (*sp == c)
                        return(sp);
        } while (*sp++);
        return(NULL);
}
```

Figure 5.1 Entire program being edited

```
static char *sccsid = "@(#)rmail.c 4.1 (Berkeley) 10/1/80";

char *index();

main(argc, argv)
char **argv;
{..
}

..

char *
index(sp, c)
register char *sp, c;
{
        do {..
}
```

Figure 5.2 Elision with focus at root of tree

```
if (argc != 2) {..

for (;;) {
        fgets(lbuf, sizeof lbuf, stdin);
        if (strncmp(lbuf, "From ", 5) && ..)
                break;
..
        sscanf(lbuf, "%s %s", junk, ufrom);
        cp = lbuf;
        for (;;) {
                cp = index(cp+1, r);
                if (cp == NULL)
                        cp = "remote from somewhere";
                if (strncmp(cp, "remote from ", 12)==0)
                        break;
        }
        sscanf(cp, "remote from %s", sys);
        strcat(from, sys);
        strcat(from, "!");
}
..
```

Figure 5.3 Elision with focus deep in tree

on the screen at the conclusion of the algorithm.[4] The second is a *priority queue* consisting of nodes waiting to be added to the display, ordered by their priority. (The priority is the distance to the focus as defined above. A large number represents a low priority.)

There are also three hash tables to speed up the algorithm. One hash table contains nodes that have already been expanded. Another contains nodes that are on the display. A third contains, for nearby utokens, values of the next and previous utokens in the fringe of the tree, speeding up the operation of searching along the fringe.

There are five phases to the priority display algorithm: *initialization*, *priority expansion*, *display fillout*, *display*, and *cleanup*. The general idea of the algorithm is first to complete the display vertically (to determine the lines of the display), and then to fill the lines out until they are full or complete. The initialization phase places the focus on the priority queue with priority 1, and puts a single elision on the display.

The priority expansion phase is the heart of the technique. In a main loop, it takes the front entry, N, from the priority queue. N is expanded by placing all neighbors (parent, immediate siblings, and children) that have not already been expanded onto the priority queue, with priorities calculated from that of N by the above rule. (A hash table is used to keep track of which nodes have already been expanded.) Then, if N is a token, and if it will fit, it is added to the display.

Even after one token has been determined not to fit, a nearby smaller token with loer priority might still fit. If the priority expansion phase were continued until the queue were empty, the entire tree would be searched before the phase could end. The algorithm in [Alberga 1979] stops when "The display is full". Our display usually does not become "full" because the possibility almost always exists of finding a token that fits.

[4]An elision represents one or more tokens, adjacent in the fringe of the tree, whose text is not displayed. Elisions do not necessarily correspond to nonterminals. Nonterminals are never placed on the display.

In order to avoid searching the entire tree, the priority expansion phase is terminated as soon as the number of lines in the display is equal to the number of lines on the screen. (The number of lines in the display can be found by counting the newlines in the tokens and elisions on the display.) This phase terminates quickly.

After the priority expansion phase, the display will typically have a number of very short lines, containing mostly elisions, with only a few tokens. There is usually room within single lines to expand many of these elisions. The display fillout phase traverses the display, from left to right, checking each token in each elision to see if it fits on that line. Those that fit are added to the display. Since the lines are traversed from left to right instead of by priority, the distance rule is violated within lines filled out by this phase. In practice, this violation does not seem to matter, because the line containing the focus is usually already filled out in the previous phase.

The next phase moves along the display list, printing each token and elision that is found. Finally, the memory used for the data structures is freed.

A more detailed explanation of some of the operations of the priority expansion phase is needed. In the following paragraphs, let N be a node, at the front of the priority queue, being considered for expansion.

If N were added to the display, it would replace all or part of one of the elisions in the display. One question is, which elision? Consider, for example, expansion (c)→(d) in Figure 5.4, in which the first "{" is added.[5] When the token "{" is to be added to the display shown in (c), there are several elisions. By inspection, it is possible to determine that the brace is part of the elision at the end of the first line. An algorithm for determining which elision is involved, however, is not obvious.

One possible method to find the elision might be to assign an elision to N when it is entered into the priority queue, calculated from the elision associated with the node from

[5]Disregard the rest of the figure for now.

which N was expanded. This method does not work, for two reasons. First, the elision so designated could be split into a number of elisions by the later addition of some other token to the display. Second, the path taken from the focus (the same path used to calculate the priority) can go up in the tree and then either branch to the left or to the right before coming down to the fringe of the tree at a far away location in a different elision.

The solution to the problem is to search along the fringe of the tree in both directions from N until a node is found that is in the display. A hash table makes it quick to determine if and where the node is in the display.

The second question is "How do we determine if N fits in the display?" This is a complex question because adding certain tokens will force lines in the display to be split. The question is answered by determining what the display structure is after expansion of N, and checking to see if this expansion fits on the screen.

There are many possible ways to add N to the display. These methods differ in the treatment of elisions. The method chosen here attempts to make the display easy for the user to understand, by keeping elisions that represent one or more whole lines on separate lines. We will use the example in Figure 5.4 to illustrate the problem. This figure shows six examples of adding a token to the display.

Consider the tree to represent the program fragment shown ("Expanded fragment"), with the priority queue ("Initial pqueue"). The initial display is the one labelled "(a)". Six expansions of elisions are shown that illustrate six different cases.[6] Note that if the bodies of the **if** and **else** clauses were longer than one line each, the example here would be unaffected, since one elision can represent several lines.

In the first expansion (a)→(b), N is the second **else**. N is removed from the priority queue. The elision after the **if** is determined to be the one containing N, and is split into

[6]To simplify the presentation, the expansions in the tree, including the priority calculations and additions to the priority queue, are not shown here.

Expanded fragment:
if (test1) {
        x = 1;
} else if (test2) {
        x = 2;
} else {
        x = 3;
}
	Initial pqueue:
	else (second)
	else (first)
	{ (first)
	{ (third)
	} (third)
	} (first)
		(a)
		if..

(b) if.. else..	(c) if.. .. .. else.. .. .. else..	(d) if.. { .. .. else.. .. .. else..

(e)
if.. {
..
.. else..
..
.. else {
..
	(f)
	if.. {
	..
	.. else..
	..
	.. else {
	..
	}
		(g)
		if.. {
		..
		.. else..
		..
		} else {
		..
		}

Figure 5.4. Example of Filling Out Elisions

three pieces: an elision, the **else**, and another elision. Since the **else** neither begins nor ends a line, the display remains one line long.

The worst case is illustrated by (b)→(c). N is the first **else**. The elision is split into seven separate pieces:

(1) The part up to the first newline. (In this example, "(test1) {".)

(2) The part between (1) and (3). ("x = 1;".)

(3) The part from the last newline before N to N. ("}".)

(4) The node, N, being expanded. ("else".)

(5) The part from N to the next newline. ("if (test2) {".)

(6) The part between (5) and (7). ("x = 2;".)

(7) The part from the last newline to the end. ("}".)

Usually several of these pieces are empty. Parts (2) and (6) might span several lines; the others are always wholly contained in one line. In this case, all seven pieces are nonempty. In this example, lines are split in the display before parts (2), (3), (6), and (7), since the first token of these elisions begins a line.

In the general case, there are six splits between the seven pieces. Four splits (before 2, 3, 6, and 7) might cause newlines to be added to the display. A newline is added to the display if the token, or the first token of an elision, has a newline preceding it. Unless a blank line is involved, no more than one newline per split is added to the display.

In the priority expansion phase, a token fits if the total number of newlines added plus the number of lines already on the display does not exceed the number of lines in the window, providing that the number of characters on the new line containing N does not exceed the width of the screen.

In the display fillout phase, only tokens on the same line as the original elision are considered (i.e., pieces (1) and (2) of the example are always empty). In this phase, a token fits if the number of characters on the new line containing N does not exceed the width of the screen. Lines are never split, leaving elisions at the ends of lines in order to maximize information on the screen.

The four other cases in the figure are added to the display in a similar manner. The expansion can be in the middle of a line: cases (a)→(b) and (b)→(c); at the end: cases (c)→(d, d)→(e); or at the beginning: cases (e)→(f) and (f)→(g). This may: cases (b)→(c), (d)→(e) and (e)→(f); or may not: cases (a)→(b), (c)→(d) and (f)→(g) cause lines to be split. A split can cause one extra line: cases (d)→(e) and (e)→(f); or several: cases (b)→(c).

5.4.4. Summary

Three display algorithms are presented here. Only the off-edge and priority elision algorithms merit serious consideration for a display editor.

The off-edge algorithm is the choice of most text editors, primarily because priority elision requires a tree to be present. Even with the tree present, the off-edge method has its advantages. It is simpler for a user to understand. It costs only about 2/3 as much CPU time to run, and examines less of the tree than the priority elision algorithm, resulting in fewer page faults. It is considerably simpler to implement. Less redrawing of the screen will be necessary as changes are made to the program.

The main advantage of the priority elision algorithm is that it is possible to see a more "macro" view of the program. A typical terminal has a screen with only 24 lines, not nearly enough for many programmers. Displaying the lines the user is most interested in (somehow allowing the user to specify which lines these are) is a desirable property of any editor.

If the user is interested in a loop with a large body, it is not possible with an ordinary text editor to get both the top and bottom of the loop on the screen at once. Text editors with multiple windows allow the top of the loop to be displayed in one window, and the bottom in another window. Such an approach is painful but better than nothing. The priority elision method allows the focus to be positioned to the first token in the loop. Since the loop beginning and end will be near the focus, in the tree, they will be favored by this algorithm. Several statements on either side of the loop will appear on the display, along with the first several inside the loop.

The Babel system makes both of these display algorithms available to the user. The default is the off-edge algorithm, because of its simplicity and lower cost. The priority elision algorithm can be easily requested when needed.

5.5. Tree Editor

A command routine is allowed to change the tree in any way it wishes, provided that errors are properly detected and handled. However, since the editor strives to present a text oriented interface, textual subroutines are important. The routine **changesection(first,last,newtext)** accepts two text positions and a character string, and replaces the portion of the tree between those two text positions, inclusive, with the new character string. (Text positions are defined in section 3.5.) Other primitive operations include **nexttextpos(tp)** and **prevtextpos(tp)**, which return the next (previous) text position relative to the input text position tp (that is, they move forward or backward one character), and **getlinetext(tp)** which returns a character string representing the textual representation of the line containing text position tp. Using these primitive operations, it is possible to build an editor that appears to be a text editor.

Many text editors attempt to provide structure commands, such as "delete sentence," "move forward to the next section" (or procedure, statement, or other syntactic unit), "check matching parentheses," etc. They are usually fooled by comments and strings. In Babel, it is possible for a structure command to determine such boundaries *correctly* by examining the tree. However, a simple tree operation, such as pruning a subtree, cannot guarantee anything about the correctness of the resulting program. Hence, a Babel structure command can *look* at the tree to determine *textual boundaries*, and should then make a *textual change* to the program through the routines described here.

Since the editor does not keep a textual representation of the file, the interface just described must be built in terms of the tree routines. This section outlines the steps used to implement the **changesection** interface. (The other routines are straightforward to implement.)

(1) The inputs are text positions *B* and *Y*, the first and last text position to be replaced, and *newtext*, the replacement text.

(2) The tokens *A* and *Z* are found[7]. These are the first real tokens (not comments or error tokens) outside the range from B to Y. They represent the one token of context required by [Ghezzi 1979] to correctly reparse with an LALR(1) parser. This also causes the scanner to be given whole tokens to rescan. The portion of the tree fringe from A to Z is the *old firewall*, which will be replaced by the newly scanned *middle firewall*.

(3) A buffer of the right size is created, and then the string to be scanned is created. There are two nearly identical steps here, counting the number of characters in the string and then creating it. The fringe of the tree from A to Z is converted to a string, replacing the portion from B to Y with *newtext*. This is done by iterating along the text positions from the beginning of A to the end of Z, accumulating characters before B and after Y. When B is reached, *newtext* is appended to the string.

(4) If a simple preprocessor were to be used, it would be applied to the string at this point.

(5) The string is passed to the incremental scanner, which returns a list of tokens, with white space and comments suitably attached. This string is the *middle firewall*.

(6) The rest of the firewall is created. Starting from A, the editor works its way up and to the left, as defined in [Ghezzi 1979] and simplified in chapter 2, until the endmarker is reached. This is the *left firewall*. Starting from Z, the *right firewall* is created symmetrically. The three pieces are linked together to form the *firewall*, a list of terminals and nonterminals, that is ready to be incrementally parsed.

(7) An *error nonterminal* is created which is made the parent of each node in the firewall (as though there were a syntax error). If checking has been turned off, or if there are commands waiting, the editor marks the firewall "dirty", and goes on to the next

[7]The mnemonic value of A, B, Y, and Z is in their position in the alphabet. B and Y are *almost* at the ends of the middle firewall, while A and Z are at the ends. (See figure 3.1.)

command. (When the editor becomes idle, if the firewall has been marked "dirty", control resumes from this point to catch up on unfinished processing.)

(8) The firewall is passed to the incremental parser. The parser returns an indication of success or failure, and a tree node, which is either the root of a successfully parsed, complete parse tree, or the token or nonterminal that caused the error.

There are now two possibilities. Either the parse succeeded or it failed.

(9s) If the parse succeeded, the tree is rerooted at the new root.

(10s) The incremental semantic evaluator is called.

(9f) If a syntax error has occurred, the firewall is converted into a list of children of a newly created parent (an *error nonterminal*) which is made the root of the tree.

(10f) The token or nonterminal rejected by the parser is marked "syntax error".

All this syntactic and semantic checking is optional. It is possible to check only syntax, or to check nothing. (Due to the representation, lexical checking is always done.) To turn off all checking, the user sets a flag either when starting up the editor, or with an editor command. In this case, only the first 7 steps of the above method are done. There is an explicit check command that will complete the final steps, in effect checking the syntax and semantics of the program.

Semantic checking can also be turned off, leaving syntactic checking turned on. In the case of a syntactically correct program, the tree will be completely rebuilt from the firewall, and the firewall is no longer valid. The record of what still has to be done (which is implicitly stored in the firewall) is lost. For efficiency reasons, in the current implementation turning off semantic checking turns off all bookkeeping related to semantic checking. The tree is marked "syntax only" and further invocations of the editor only check syntax. This mode is quite suitable for a program that the user only wishes syntax checking on, or for which the language description only defines syntax.

5.6. Incremental Scanner

The incremental scanner used in Babel is quite similar to the scanner that **lex** [Lesk 1979] generates. There are a number of differences, all very minor.

The scanner is incremental in that it is possible for only a portion of the program text to be passed to it. The calling routine must ensure that the text passed does not separate a token; that is, if any part of a token is in the text, the entire token is in the text. Since comments are viewed as part of the token they precede, any comments before tokens in the text must also be present in the text. The editor must generate text to be rescanned, but this is not difficult to generate since the information is in the tree. In order to be completely driven by external tables, the lexical specification tables, generated by **Idlproc** and **lex**, are read in from a file instead of being compiled.

Since it must be possible to reconstruct the textual representation of the tree for display, or for an incremental scan, comments and white space cannot be thrown away. The tables include special tokens **Comment**, **Whitespace**, and **Id**. At the lowest level, these tokens are scanned like any other token. At a higher level, comments are attached to the ordinary token that follows them. White space is reduced to a pair of integers designating the newline count and blank space count. The pair is attached to the token or comment that follows. Unrecognized input (scanner errors) are made into *error tokens* and treated like comments. Identifiers are checked against a reserved word table (also read from the file) and converted into the appropriate reserved word token if found.

Since long comments are often present in programs and are one of the most common reasons cited for not using **lex**,[8] they must be handled specially. Any comment internally spans only one line. A comment that is longer than one line is **broken** up into multiple one line comments by the scanner. The textual representation is unchanged, but internally the

[8] **Lex** keeps the text of the current token in a static buffer. A long token, such as a big comment, will overflow this buffer. Modifications to the **lex** canned scanner to avoid overflowing another static buffer were also necessary.

long comment is treated as several comments. This prevents the usual problem of overflowing buffers with fixed size. Although the data structure could represent a situation with a token between two parts of a comment, such a situation cannot arise because the user has no commands for manipulating the data structure at such a low level. If the user attempted to insert a token into the middle of a comment, the token would become part of the text of the comment rather than a token.

Normally, a scanner is called once for each token a parser needs. Because of the possibility of syntax errors, this scheme is unsuitable for an incremental environment such as Babel. Instead, the scanner is called repeatedly until the input is exhausted, and the tokens returned are linked together in a linear list. When the parser needs a token, a *token server* is called which merely takes one token from this list. Since the list is left intact, it is easy to reconstruct the input in case of error. Since token nodes must be generated anyway for placement in the tree, and since these nodes are tree nodes that do not all have to reside in primary memory, there is no memory penalty for this prescanning. The same incremental scanner is used for the **bparse** tool, which scans the entire program before parsing any of it.

5.7. Incremental Parser

The incremental parser used here is based largely on [Ghezzi 1979]. The simplifications made are noted in section 2.10. Improvements based on a later paper [Ghezzi 1980] seem possible and are planned for a future version.

Tables generated by **yacc** are read in from the language file. A modified version of the yacc parser is used for incremental parsing. The editor is responsible for arranging that the firewall is linked together in such a way that the token server will return the correct next input to the parser.

The incremental parser is responsible for creating a correct tree from the old tree and the firewall. Since the firewall is made up of both tokens and nonterminals, the incremental parser must be modified to accept nonterminals from the scanner as well as tokens. It turns out that this is easy to do.

When a symbol is accepted by the parser, if it is a nonterminal that is legal in the current state, control is immediately transferred to the reduction portion of the parser. The nonterminal returned from the token server is used in place of the nonterminal node normally built in the first part of the reduction. Then the nonterminal and production it came from are used to determine the new state to enter. In the case of a reduction by an empty production, rather than creating a new empty nonterminal, if the lookahead symbol is the same nonterminal, the lookahead node is used.

A representation stack[9] is kept in parallel with the stack used by the parser. An element of the representation stack contains a subtree corresponding to the symbol on the parser stack.

Reductions are handled by the parser; the same processing is done for all productions. When a reduction is made, the production number and nonterminal number are easily determined from the tables. A new nonterminal tree node is created containing these two numbers. The tree nodes that are popped from the representation stack become children of the new tree node. This new node is then pushed back on the representation stack. The usual reduction is made with the parser stack.

The parser used in Babel is based on the **yacc** parser and not on an ordinary LR parser. Some extra effort was expended in this implementation to compensate for the compression of the **yacc** tables. Since the **yacc** tables do not distinguish between a default state and an error state when a nonterminal is found, error detection is more difficult. The first token of the nonterminal must be examined, to determine if it would be accepted. For some grammars, this ambiguity may cause an incorrect program to be accepted, but it has not been a problem with the languages implemented. This problem does not arise with normal LR tables.

[9]The representation stack is similar to a semantic stack, but the information kept on the stack is syntactic in nature.

5.8. Incremental Semantics

For most languages, context-free grammars can not sufficiently describe all the language restrictions. Those restrictions requiring context-sensitive checking are called *semantic restrictions* or simply *semantics*. Babel includes a formalism to check for semantic errors, based on the notion of attribute grammars, and does incremental checking of syntactically correct programs for such errors.

Chapter 2 includes a description of the attribute grammar formalism, and the previous work by Reps on incremental attribute evaluation. This section discusses the Babel contributions, implementing incremental attribute evaluation in an incremental parsing environment, allowing shared symbol tables, and solving the deletion problem.

A Babel language description, written in LDL, will have a number of semantic rules for each syntactic production. Each rule is an evaluation function, written in a dialect of Lisp, for an attribute in the production. The rule also lists the attributes which are used and which are set, so that Babel does not have to interpret the Lisp code to determine what is used and what is set. (See chapter 6 for a more detailed description of LDL.)

The LDL processor, in translating the LDL description to internal tables, will form for each production p, the dependency graph $D[p] = (V,E)$ where V is the set of attributes of all symbols in p, and $e = (v1,v2) \in E$ iff v2 depends directly on v1. This graph shows the attribute interdependencies within the production. The graph is stored among the tables in the language intermediate file, along with the Lisp functions.

Babel views symbol tables as linked data structures made up of smaller *symbol table building blocks* (STBB). STBB's may contain any number of identifiers, but any one identifier can only appear once in an STBB. With each identifier is associated a fixed number of *attributes*. Each attribute can be either an integer or a tree pointer. (Character strings, other than the identifier itself, can be represented by storing a tree pointer, referencing a token node whose text is the desired character string.)

At any location in a program, there is usually a "current symbol table" used to look up identifiers. Such a symbol table is implemented as a linear list of STBBs. The list specifies the scoping rules of the language. (For example, in an Algol-style block structured language, each STBB would represent one nested block in the path to the outermost block.) Since the scoping rules may specify different orders of preference at different locations in the program (e.g., Pascal **with** statements, Ada imports) it is possible to link the STBBs into different orders in different places in the program. See section 6.1.4 for a description of the notation used to specify these rules for a given language.

5.8.1. Solutions to Attribute Grammar Problems

In this section, solutions are presented to the problems outlined in chapter 2. First, it is shown how to adapt Reps' methods to incremental LR parsing. Second, detecting and avoiding circular grammars is discussed. Third, it is shown how to handle the scoping rules of typical languages. Fourth, a method for sharing symbol tables in an incremental environment is given.

First, we define the *nondestructive sharing property* to be the property that if *A* is the set of attributes that are already evaluated and in place in the tree before the evaluation of an attribute *a*, then after *a* is evaluated, each attribute in *A* retains the same value as before the evaluation of *a*. This property is necessary in an incremental environment, to cause the effect of any evaluation to be "undone" by ignoring the result of the evaluation.

5.8.1.1. Incremental Attribute Evaluation with Incremental Parsing

When the parser has reparsed the portion of the tree above the firewall, all the attributes above the firewall must be reevaluated. In addition, any attributes below the firewall that directly or indirectly depend on attributes above the firewall must be reevaluated. This process is similar to that in [Reps 1981a] except for the nature of the change to the tree.

The Reps methods cannot be applied directly because Babel changes are not subtree replacements. Rather the portion of the tree above the firewall is completely rebuilt (using existing nodes where they are identical) and the new tree is rooted in place of the old. It is not possible to view this as a subtree replacement at the root because the resulting tree will not be prepared for propagation. (The attribute values do not already exist above the firewall.)

The solution to this problem is to start with a larger attribute dependency graph M, representing attributes of all nodes above the firewall.[10] This leads to an immediate simplification. Since M can only expand downward, the characteristic superior graphs are never needed.

A node in M must be created for each attribute of each node above the firewall. Since there is always exactly one reduction for each node above the firewall, the obvious time to add to M is in the reduction. Unfortunately, M must be created top-down to avoid extra path nodes in the finished graph, and the reductions are made bottom-up. For this reason, the initial value of M is created in a two step process. In the first step, for each reduction, the tree node is pushed on a stack of nodes to be placed on M. The second step occurs after the parse has been completed. Each node on the stack is expanded (as though an evaluated attribute had changed value) and the resulting graph added to M.[11]

The remaining problem is how to keep the characteristic subordinate graphs available when they are needed (Reps' requirement of keeping the tree prepared for propagation at the cursor). Since the parser is bottom-up, nodes are always created as parents of previously created nodes, never by changing nodes from underneath an existing node. Thus once a node is created, the characteristic subordinate graph will never change. The creation of a characteristic subordinate graph, of a node, N, requires the dependency graph D[p] of

[10]See section 2.12 for the definition of the partial attribute dependency graph M.

[11]Note that the order of expansions on the stack is the exact reverse of the bottom-up reduction order. This is not, in general, the same order that would be produced by a canonical top-down parser. However, this does not matter, the only requirement here is that a node's parent be expanded before the node itself.

the production p, of which N is the left hand side, and the characteristic subordinate graphs of all nodes which are its children. These are all available at the time of the reduction forming the node, and since the node is formed at that time, it is not possible to form the characteristic subordinate graph earlier. For these reasons, the characteristic subordinate graph is determined at reduction time and stored in the tree.

5.8.1.2. Circularity

Babel cannot handle circular grammars since there is no evaluation order which permits all arguments to be evaluated before the nodes themselves. A circular grammar is detected during the incremental semantic evaluation phase. If M is nonempty, but there is no node with in-degree zero in M, the grammar is circular. It is possible, using an algorithm given in [Knuth 1968] to detect circularity in the preprocessor. Since circularity is trivially detected at runtime, and since attribute grammars which are accidently circular seem to be rare in practice, the exponential cost of this algorithm, as discussed in [Jazayeri 1974], may be excessive. The current prototype implementation does not check for circularity in the preprocessor. It would be possible to implement a circularity check, but an option to turn off this check would be useful in the case of a grammar that is known to be noncircular.

When constructing a symbol table, it is often convenient to have cycles in the data structure. For example, in a Pascal enumerated type declaration, it would be useful to be able to tell, not only for an enumerated type name, what the constants of that type are, but for a given constant, what the type is. Creation of such a symbol table using the attribute grammar formalism will either violate the nondestructive sharing property, or cause a circular grammar. Babel symbol tables must be carefully constructed to avoid such cycles.

In the case of Pascal enumerated types, circularity is avoided by noting that being able to find the constants of a type is not necessary in standard Pascal. Extensions to this language feature, such as being allowed to write values of enumerated types, or the 'FIRST and 'LAST attributes of Ada[12], which seemingly require a pointer from the type to the members of the type, can be handled in Babel, with some extra work, because the evaluation routines are allowed to inspect the tree nonlocally and save additional information. In this case, the names of the first and last elements of the enumerated type can be stored in the symbol table *type* attribute.

It is also possible to construct an arbitrary *code created data structure* in the tree, allowing cycles, without using attribute values to refer to portions of the data structure. A single reference to the entire data structure would be placed in an attribute value. Careful symbol table design will probably make this unnecessary for reasonable languages. Since pure Lisp is universal, however, the power exists to have a description with a single attribute attached to the root, and an evaluation function which examines the tree, builds arbitrary symbol tables and other structures, checks for errors, and so on. Such an implementation is not intended, since it does not make use of any of the incremental semantic checking features of Babel, but it shows that the power to handle languages requiring circular grammars does exist in Babel.

A difficult case to handle in a real language is the Pascal enumerated type. Normally, the type attribute in a type declaration is synthesized. However, in the enumerated type, it is essential to know, for each of the constants, what the type is. This would seemingly require the type attribute to be inherited. It cannot be both. However, in each instance of a type declaration, the type is either enumerated or not enumerated, so it is possible to specify a non-circular grammar. One solution is to pass the name of the type (which must be part of the symbol table entry for the type) to the bottom of the list of constants, where the type is created. Another solution is to restructure the grammar so that type is either enumerated or otherwise, and treat them differently at the top level.

[12]These attributes allow the programmer access to the first and last elements of an enumerated type, without referring to the name of the element.

Another interesting case is a pair of mutually dependent data structures. For example, in Pascal, it is possible to have two record types **a** and **b**, where **a** contains a pointer to a **b** and **b** contains a pointer to an **a**. This case can be handled without resorting to code-created data structures by observing that Pascal requires a restricted syntax for pointer types:

<type> ::= ^ identifier

It is only necessary to store the name of the identifier in the symbol table, not the type. The identifier can be looked up and checked when it is used. (On most machines, pointers are the same size, so storage can be allocated without knowing what type the identifier represents, should the language implementor choose to do storage allocation in Babel.)

5.8.1.3. Scoping

Algol style block structure is easily handled by Babel. The system provides *symbol table building blocks* (STBB) for symbol tables which are lists of identifiers and *symbol table attributes* (STA).[13] Each STBB represents one local scope. Normally, each entry in an STBB has a unique name. (In the case of a language such as Ada that allows overloading, the language implementor would write routines, in Lisp, to do lookups and resolve ambiguities as needed.) Each STBB has one extra field, called the *global pointer* **gp(stbb)**. This field references the STBB of the next outer scoping level.

There are two primitive functions to look up an identifier. **Lookone(stbb,name)** looks for the name in the given STBB and returns a reference to the entry found, or a failure indication. **Lookup(stbb,name)** is the same, except that if no matching entry can be found, an attempt is made to find *name* in the STBB referenced by the global pointer **gp(stbb)**. If that fails, the next scope **gp(gp(stbb))** is tried, and so on until a match or a null global pointer is found.

[13]STA's are not to be confused with attribute grammar attributes. STA's are merely information placed in the symbol table for each identifier, such as the type of a variable.

Lookup is used for ordinary identifier lookups. Lookone is intended to be used in a declaration rule to check for multiple declarations. While it would be possible for the system to check automatically for multiple declarations when an identifier is added, this responsibility was left to the language implementor, both to provide extra flexibility, and to allow a better choice of error messages and locations to attach the messages to in the tree. This extra flexibility would be useful, for example, in handling overloading.

This scheme is sufficient to handle ordinary block structured languages, and more. For example, in Pascal, each record type invokes a separate scope for the fields of the record. It is possible to use an STBB for each record type. The global pointer field can refer to the ordinary symbol table for scoping purposes, or it can be null, allowing the language implementor to write a routine to check record scopes that are open (from **with** statements) before the normal symbol table.

Additional power is attained by the addition of the **reglob** primitive, which changes the value of the global pointer field of an STBB. This change is done nondestructively by copying the top node of the STBB with a new global pointer field. This permits scopes to be linked together in arbitrary orders, which can differ in different parts of the program. Such power is especially useful for the Pascal **with** statement, since dynamic relinking of this kind is needed to handle cases such as

```
with a do with b do x := y;
with b do with a do x := y;
```

where both a and b are records with fields y.

The universality of Lisp and the ability to use code created data structures shows that any language semantics can be handled. In this respect, semantic checking is one of the strongest aspects of Babel, since there are restrictions on the lexical and syntactic rules which rule out several existing languages. Since a syntax-only LDL description of Ada already exists, a full semantic checking implementation of Ada is therefore possible.

5.8.1.4. Sharing Symbol Tables

The sharing of symbol tables is solved in the Babel system by making fundamental use of the tree structure. Babel STBB's are linear lists, made up of two kinds of tree nodes: *SYMTAB nodes* representing all or part of an STBB, and *STE nodes* representing one entry in a symbol table. Symtab nodes contain a global pointer, a pointer to an STE node, and a pointer to another SYMTAB node. An STE node contains the name of the identifier and an array of attribute values. An STBB consists of a list of SYMTAB nodes and STE nodes, linked together as shown in figure 5.2. (The SYMTAB nodes are referenced by pointers pa and pb. The STE nodes are labeled A and B.)

The convention that a SYMTAB node with no children represents an empty STBB is adopted. A null pointer cannot be used for an empty STBB, because null STBB's must still have global pointers.[14]

This list arrangement has the property that adding an entry to an STBB will not invalidate existing references to it, nor will it change them. For example, suppose that the STBB referred to by pb in figure 5.2 exists, and an entry for A is added, referred to by pa. The reference pb will continue to refer to an STBB containing B and lower entries. The reference pa will refer to an STBB containing A, B, and lower entries. If the user later changes

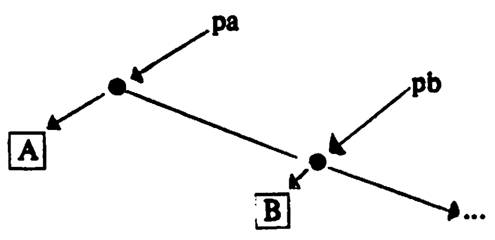

Figure 5.2 Symbol Table Building Block

[14]Algol-like languages where begin-end can bracket either a block (with a symbol table) or a compound statement (which is not a scope and has no symbol table) are not a problem because the difference is resolved syntactically.

A to C in the program, a new STE for C will be created, a SYMTAB node will be added above the C and pb nodes, and an STBB will exist containing C, B, and lower entries, without A. (See figure 5.3.) Thus, the nondestructive sharing property has been preserved.

The obvious problem with the above method is that linear lists make inefficient symbol tables, due to the linear search time. Since each of these nodes is a tree node, with a potential disk access for each reference, a search down a linear list is even more expensive than a linear search in memory. Sorting the list is not possible, because the nondestructive sharing property requires that the order of the symbol table be the order of the declarations in the program. Restructuring the STBB into a binary search tree is also not possible since binary search trees add to the bottom of the tree rather than the top.

The solution to this problem is transparent hashing. A third kind of tree node is introduced, called a *hash table node*. Such a node contains a hashed symbol table for the entire STBB, containing for each identifier a pointer to the STE node for the identifier. The language implementation is expected to call the hash function with a completely built symbol table, and pass the result as a symbol table to the remainder of the program. The lookup functions accept either a hashed or unhashed STBB, so that partial symbol tables can be used in the declaration portion of the program.

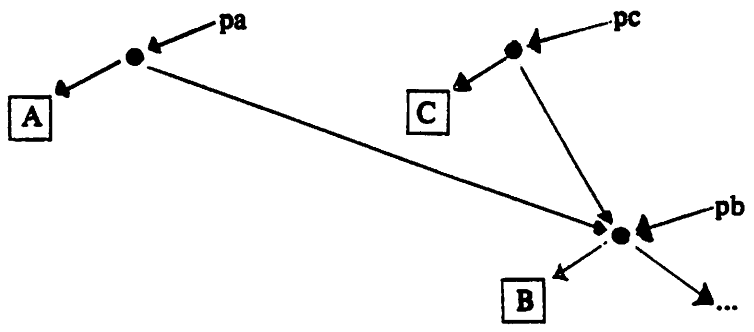

Figure 5.3 Symbol Table Building Block

In practice, this kind of symbol table works quite well. For very simple languages, it is very straightforward to produce an attribute grammar building up such a symbol table. For real languages such as Pascal, it requires care. Symbol table construction is easier in a compiler than in Babel, because the symbol table is global, and any declaration which must add to the symbol table merely adds to the global symbol table. When using an attribute grammar, it is necessary to pass a partially formed symbol table down to any nonterminal that *might* need to add to it, and to accept back a potentially augmented symbol table to pass on up the tree.

For example, the Pascal nonterminal **type**, which intuitively should be passed an inherited symbol table and return a synthesized type, which has been looked up in that symbol table, must also return a synthesized symbol table, since the type might be a record, or an enumerated type. Thus, any nonterminal which can derive **type** must be passed a symbol table, and the resulting synthesized symbol table must be accepted back from the type.

An alternative structure to the linear list implementation used here, as suggested by Reps [Reps 1981b] is to use a 2-3 tree. The advantage to the 2-3 tree are that hashing is no longer essential (since lookup time can be done in $O(\log n)$ time where n is the number of entries in the symbol table instead of $O(n)$ time. This makes lookups during the creation of an STBB faster. The disadvantage is that extra tree nodes will be created and discarded while building a tree. Also, if no hashing is done, lookups in the body of the scope will take $O(\log n)$ time instead of constant time. The decision is essentially a space/time tradeoff.

5.9. Preprocessors

There are a few languages which use a preprocessor which is run before the scanner in order to augment the language. Fortran preprocessors are common. Assemblers have used macros for years. Other languages using preprocessors include C, Bliss, and PL/I.

Such preprocessors have three properties in common: (a) they drastically extend the programming language, (b) they are simple, *ad hoc* tools requiring little effort to implement, and (c) they can usually be implemented in one pass.

5.10. Problems

Preprocessors make the job of a language editor considerably harder. To see why, consider the operation of one preprocessor, the C preprocessor **cpp** [Kernighan 1978]. **Cpp** is a filter. It accepts one text file as input and produces another text file as output. The output text file is the result of applying preprocessor transformations on the input. Macros are expanded, comments are stripped, header files are textually included, conditionally compiled code is removed, and other such drastic changes are made. Such changes are not readily invertible, unless extra information is kept for reconstructing the source text.

Since the programmer prepares input for the preprocessor, it appears that a language editor should not edit the language accepted by the compiler, but rather the language accepted by the preprocessor. But what is this language? There is no context free grammar to describe it. Macros allow significant syntax extensions. For example, it is possible to define macros called **if**, **then**, **else**, **elif**, and **fi** to create an Algol 68 style conditional statement with a mandatory **fi**.

It is possible to handle some preprocessor features by placing an incremental preprocessor before the incremental scanner, on a line-by-line basis. Before each line is passed to the scanner, it is passed through the preprocessor, which can turn it into any number of lines of output.

This approach has a number of problems, however, and is not powerful enough to handle all the features of **cpp**. The worst problem is the nature of **cpp** scoping, which was designed with one-pass, batch translation in mind. In **cpp** the scope of an identifier is from the point of definition in the input file, either to the end of the file, or to an **undefine** line, which removes the identifier from the symbol table at that point. The symbol table has a new set of values after each addition or deletion. This convention divides the input file into a number of zones, each of which has a different set of values in the symbol table. There are no blocks. Overlapping of scopes does not imply nesting. Identifiers can be defined and undefined in any order.

In order to faithfully implement these semantics, it would be necessary to maintain a symbol table for each zone. The overhead caused by this requirement is excessive, and violates the spirit of the original tool, to be as simple as possible.

5.11. Subsets

Approximations to the semantics of **cpp** are possible. In one possible subset, the **define** command is allowed. The **undefine** command is forbidden. All definitions of macros must appear before any uses. (Not only must they appear *textually* before, but chronologically before also, so that uses will expand properly.) This permits the straightforward maintenance of a single symbol table. Passing a **define** line through the preprocessor adds to the symbol table. Passing a text line through expands any macros involved. In addition to the restrictions listed above, this approach is unable to detect multiple definitions of the same macro, since they are indistinguishable from rescanning of the same macro definition. It is not possible to *change* an existing macro definition, since this violates the chronological restriction.

The chronological restriction can be dropped if, upon receiving a macro definition, the implementation scans from the point of the definition, throughout the entire tree, looking for instances of the token defined, and reprocessing lines containing that token. An incremental scan, parse, and semantic check would be required for *each* occurrence found. (It is possible to prevent more than one parse and semantic check, in effect pretending all parses but the last caused a syntax error.)

The positional restriction can be dropped by making effective use of Babel linear symbol tables, without hashing. To the first token on each line is attached a symbol table pointer, representing the symbol table at that point in the program. Moving closer to the end of the file, as new macros are defined, new entries are added to the current symbol table, and the resulting symbol table is attached at that point in the tree. The nondestructive sharing property assures that earlier lines will continue to have their less complete version of the symbol table.

Finally, the restriction forbidding **undefine** can be dropped by borrowing a technique from **termcap** [Horton 1980] which has special *cancel entries*. An **undefine** command cannot in general delete the macro definition from the symbol table because the definition might be in the middle of the table, and its deletion would violate the nondestructive sharing property. Instead, a cancel entry is *added* to the symbol table. The lookup routines are programmed to return failure upon encountering a cancel entry for the given name, without looking any further. The success of this strategy depends on the linear organization of the preprocessor symbol table. Unfortunately, such an implementation requires substantial amounts of reprocessing when any macro definition is changed. All later preprocessor lines in the program must be reevaluated, as well as invocations of the macro just changed, and extreme care would be required to avoid scanning, parsing, and checking semantics of most of the remainder of the program.

In addition, the problem of deletion of a macro, or changing its name, must be addressed. The solution used for semantics would cause complete reprocessing of the rest of the program.

5.11.1. Display

The display is another problem. The display algorithms work from the tree representation. Since the tree will contain expanded macros, a display from it will look very different from the text the user typed in. Similarly, if a change is made, the input to the preprocessor will be the expanded text generated from the tree instead of the original macro invocation.

One approach is to attach the invocation text to the first token of the expansion. Each token of the expansion is marked "expansion" with a pointer to the first token. The display algorithms check for such marks and produce the invocation instead of the expansion. Likewise, the tree editor must produce invocations instead of expansions as input to the preprocessor.

Another approach is based on a textual representation instead of Babel's tree representation. The text file would be stored as an ordinary text file. An auxiliary file would contain the tree. The display would be based on the text file, and all input would come from the text file instead of reconstructing the tree.[15] This approach would miss some of the advantages of Babel, such as priority elision, keeping only one copy of the file, and the nonsequentiality of the tree file, but is an avenue well worth exploring.

5.11.2. Additional Preprocessor Features

There are other features provided by **cpp**, which can be handled with some extra work. The **include** command, which includes a text file at that point in the program, is not hard to handle. It is treated as a macro, expanding to the contents of the file.

The conditional compilation feature is hard to handle satisfactorily. An ideal solution would check both possibilities and ensure that there are no errors whether the code is included or not. (In the general **cpp** case, there are two sections of code, one to be included if a condition is true, one if false. The program can omit one of these sections.) Such a solution would require merging of two separate parse trees into one, or two separate copies of the parse tree. Since multiple conditions are possible, any number of separate or merged parse trees might be needed. Since the text need not represent a single branch of a parse tree, no simple merging solution allowing multiple children of a node representing the same production is sufficient.

[15]Such an approach is being taken in [Morris 1981].

A more practical solution is to evaluate the conditions and assume that the condition values are correct. The false part is stored textually as part of the invocation, and is not checked. Some conditions can be determined by the contents of the file itself, but it is common practice to define conditions externally by parameters to the C preprocessor. This solution would require that the user define the conditions upon invoking the editor, just as is done for the preprocessor. Some kind of facility for storing initial definitions in the file would be necessary, since the user would frequently forget to define them, and in any case, it would be tedious to enter the definitions manually upon entering the editor.

A multi language preprocessor solution is a very hard problem. If there is one common property of existing preprocessors, it is that they are largely *ad hoc* Models such as context-free grammars, attribute grammars, and block structured symbol tables do not describe the preprocessors well.

The Babel system does not implement a preprocessor. Instead, a few simple provisions for the C language have been made, in order to make it possible to use Babel to edit C programs. These provisions do not handle the entire C language, but it is possible to get some checking from a large class of simple C programs.

In Babel's C description, lines beginning with the **cpp** attention character "#" are treated as comments, and the most common C macro, **FILE**, is defined as a reserved word, being treated by the grammar as any other built in type in C. Thus, programmers who make only simple use of **cpp** for manifest constants and in-line expansion of functions, and who avoid the C **typedef** construction[1], can use Babel. Note that only syntactic checking of C is being done. Macros are not checked (except syntactically as the procedure calls or variables they appear to be) and semantics are not checked. This simple solution will not

[1]The **typedef** construction in C defines a new type. For example, "**typedef char** *charptr" defines a new type, charptr, which is a pointer to a character. C compilers implement **typedef** by making a new reserved word, **charptr**. The dynamic addition of reserved words violates the static lexical requirements of Babel. An implementation allowing an identifier where a type is allowed would have an ambiguous grammar. For example, the block "{ a * b; }" could be either an evaluation of the expression a times b, or a declaration of b as a pointer to an a.

work with semantic checking, because macros need to be expanded to determine correctly the types of the expressions to which they expand.

5.12. Prettyprinting

This section describes the prettyprinting facilities in Babel. An offline prettyprinter is described, and it is shown why the program is not prettyprinted as it is entered.

The LDL language description contains information showing how a program should be formatted. This information is not an inflexible rule, but rather a suggested way to format a program. The Babel system allows a user to format his program as he sees fit, just as a text editor would. It is often useful in practice, however, to be able to reformat a program which has been somehow messed up. A large class of tools to do such reformatting exists; such tools are called *prettyprinters*.

It is quite easy to prettyprint a Babel program, since the tree structure of the program is already there. Pretty printing can be done without having to guess whether a particular keyword, such as **begin**, is a keyword or part of a comment. In a situation where one instance of **begin** should cause a right shift, but another should not, if the two instances can be distinguished by the syntax of the language, it is easy to handle both cases correctly.

Babel prettyprinting information places extra pseudo-tokens in some of the productions of the grammar. There are three possible pseudo-tokens: **%nl**, **%lshift**, and **%rshift**. These directives indicate a recommended place for a newline, a shift of the left margin to the left, and a shift of the left margin to the right, respectively.

5.12.1. Off Line Prettyprinter

The Babel prettyprinting algorithm is quite simple. An indent counter is set to zero. The prettyprinter recursively walks the tree, keeping track at each level of which production it is in. When a shift directive is encountered in the current production, the indent counter is incremented or decremented by one tab stop. (The size of the tab stop can be a parameter.)

When a newline directive is encountered, the algorithm examines the next token. If the token has no newlines, a newline is inserted at the current indenting level. If the token has one or more newlines, it is left as is, but the indenting level is adjusted.

When a newline that was not expected is encountered, if a comment or blank line precedes or follows the newline, the indenting level is adjusted to the current level, otherwise, the newline is deleted.

By using the same tree library used by the other Babel tools, it is easy to write such a prettyprinter. The tree is prettyprinted in place. A change to the newline or white space count of a token (the only change that must be made by the prettyprinter) is done by copying the token into a new tree node, adjusting the value of the new node, and replacing the old node in the tree with the new node. This ensures that the prettyprinter is just another Babel editor command, saved in the history list, and can be undone with the editor **undo** command. This prettyprinter has not yet been implemented.

5.12.2. Automatic Prettyprinter

An initial goal of this project was to use this same language dependent information to prettyprint the program automatically as it is being typed in. In an LR parsing environment, we are unable to do this. This section discusses the problem.

A simple solution appears to be the following: Let B be the line being typed in, and A be the line immediately before it. The method starts at the first token of A, noting its indenting level. The tree is walked from that point to the beginning of B, counting shifts in the grammar. The indenting level of B is set to the level of A, adjusted by the number of shifts encountered from A to B.

The problem with this method is the presense of syntax errors. If the program were always syntactically correct, this method would be possible. If syntax errors are present, some of the structure of the tree is gone, and shifts will be missed.

In practice, programs almost always contain syntax errors as they are being entered, because they are usually incomplete. In an LR parsing environment, the parser can not tell which production it is accepting until it has read all of that production, making it impossible, in general, to prettyprint based on the incoming partial program.

There are a number of semi-automatic prettyprinting aids present in text editors, and any of these could be put into the user interface. **Vi**, for example, has a mode which indents the line being typed to the same level as the previous line. The user can manually adjust this default either to the right or to the left.

In Babel, the lexical structure of the program is available, even in the presense of syntax errors. This property makes such techniques as counting **begin** and **end** keywords possible. While this method is less powerful than the syntactic method outlined above, it is more powerful than simple character counting (which is fooled by comments and strings) or than the semi-automatic method of **vi**.

Finally, it is worthwhile to note that the offline algorithm above can be applied to any subtree of the program, initializing the indenting level to that of the last line before the subtree, or to a parameter entered by the user.

5.13. Error Messages

This section describes the facilities in Babel for handling error messages. Primitives for attaching error messages to the program are defined. The user interface is discussed.

Many systems simply produce error messages on the terminal and forget them. Since one of the aims of Babel is to remind the user of what still needs to be done, this approach was deemed unacceptable. Instead, error messages are kept, as a property of the program, until the error has been corrected.

Babel takes the view that error messages are character strings, attached to a node in the tree. A special type of node is used to hold error messages. This node contains the text of the error message and a hashed value of the error message for speed in comparison.

5.13.1. Implementation

Babel contains the following primitive error routines.

set_error(node, string)

Add an error message to a node. If the message is already there, nothing is done.

clear_error(node, string)

Remove an error message from a node. If there is no such message attached to the node, nothing is done.

check_error(condition, node, string)

Either **set_error** or **clear_error** depending on the result of evaluating a Boolean expression.

find_error(node)

Return the error message, if any, attached to the node.

This framework permits a routine that checks for an error to insert a one line assertion that automatically takes care of setting or clearing the error condition, such as

check_error(type(a) **=** **type**(b), node, "type clash");

Such a framework is convenient for making semantic checks. The **check_error** routine is called directly by the Lisp **check** function.

For syntax errors, the tree editor uses the **set_error** and **clear_error** routines. When a syntax error is detected, the token or nonterminal rejected by the parser is marked "syntax error" by **set_error**. When a parse successfully completes, all the nodes on the firewall are cleared of the "syntax error" message by **clear_error**. Since all nodes previously rejected by the parser will be in the firewall, clearing the firewall insures without traversing the entire tree that there will be no remaining nodes marked in error.

5.13.2. User Interface

When the tree is displayed, any token that is a descendent of a node marked in error will be highlighted. In order to find out why a node was marked in error, the user can position the focus to the node and enter the **why** command. The first error message attached to that node will be output on the message line. If none can be found, ancestors and descendents are checked.

Another approach would be to display the text of the message as soon as it is encountered. Such an approach was rejected for the following reasons. First, allowing multiple error messages on the terminal screen would clutter up the display. Second, it is common for a user to make a multiple step change to a program, going through incorrect states. Displaying messages during these states would be distracting to the user. Third, the time taken to output a number of error messages to the terminal over medium speed (1200 baud) telephone lines would be excessive.

It would be possible to list *all* error messages attached to a node, its ancestors, and its descendants, either one at a time on the message line, or by clearing and redrawing the screen. In order to avoid the cascading error message problem, only the *first* error message is output, since it is likely that the first error caused the remaining messages. If there were additional true errors, they will become apparent when the first error is corrected.

5.14. Performance

This section discusses the performance, in terms of time and space, of the current, experimental Babel implementation, and some possible improvements to Babel to improve the performance of the system. Although the system was designed with performance in mind, it was not known in advance what the trouble spots would turn out to be. Some ideas are set forth indicating how improvements might be made to future versions.

Since Babel does considerably more than a text editor, it is reasonable to expect that it would cost more to use. This expectation has proven to be true. The current experimental editor has had considerable tuning of the major performance bottlenecks that were straightforward to tune without making major changes to the structures. Performance, however, was not the overriding consideration in the construction of the system: rather, showing that such a system can work was the major goal.

Although it has turned out that making changes to a program is several times more expensive than with a text editor, moving the focus around is just as fast. Unlike the approach of [Wilcox 1976] and [Morris 1981] where moving the cursor implies recompilation of text moved over, or [Reps 1981a] where moving the focus requires recomputation of characteristic graphs, moving the focus in Babel requires only a very quick table lookup to find the correspondence between the screen and the tree, and a redisplay if the focus has been moved off the screen.

5.14.1. Time

The current implementation of the Babel editor costs varying amounts more than a text editor to run. Measurements were made of several text editors, and of Babel with various amounts of checking. Three tests were made, using an 60 line Pascal program: (1) the editor was entered and immediately exited, (2) an assignment statement was added, and (3) a declaration was added. The Unix **time** command was used to measure the CPU time needed by the editor invocation. The CPU time needed for entry and exit was subtracted from the others to arrive at the time needed to make the change. The tests were repeated to ensure that a system fluke did not throw off the numbers.

Running **be** with all checking turned off required from about 4 to 5.5 times the CPU time needed by **vi** to make a single change to the program. The difference can be explained by the number of times the screen had to be redisplayed, an operation that is more expensive in Babel than in most text editors because the tree must be examined.

(The executable statement was added further down in the program than the declaration, causing an extra screen redisplay.)

Running be with syntax checking only, the cost was about 4.5 to 6 times the cost of vi. The scanner typically rescans fewer than 10 tokens, and the size of the firewall is typically in the 20 to 30 range. Clearly, the cost of syntax checking is negligible.

Semantic checking costs more. When the executable statement was added, a ratio of 14 to 1, compared to vi, was found. Addition of a declaration required reevaluation of the entire scope of the declaration, resulting in this case in a ratio of 62 to 1, compared to vi, but the ratio will in fact be proportional to the size of the scope. The cost of reevaluation of the entire scope in the case of Babel's Pascal implementation is about one second of CPU time per line of program.

Some effort to cut down on such expensive evaluations is clearly needed to make semantic checking practical. One approach is to cut down the constant of proportionality. Another is to "batch" the evaluations to occur less often. A third is to reevaluate only the uses of the variables affected by a change to a declaration, rather than the entire scope.

There are many ways to cut down on the constant. The Babel system spends much of its time in routines to examine the tree structure. Improvements to the tree library to make tree access faster is one possibility. Better virtual memory facilities from the operating system, for example, would save the system from checking (1) if the page table for that tree node has been created, and (2) if that page needs to be read in from the disk file, two tasks that are required for each tree access. Techniques discussed below for making the tree smaller would also make it faster, since more nodes would fit in one page, fewer pages would need to be touched, causing fewer page faults. Cutting down on the number of nodes, by using an abstract tree, would cut down significantly on the number of evaluations. Finally, the attribute routines are written in interpreted Lisp, and appear to be expensive to evaluate. (About 10 attributes per second can be evaluated by Babel.) Certainly, the special **cp** case (see section 6.1.4), which accounts for over half the evaluation rules of the grammar, could be made special at runtime.

Another approach is to prevent multiple evaluations of entire scopes. Evaluations are deferred if there is another command being typed by the user. It is also possible to turn checking off and on manually. When checking is off, an explicit **check** command can be entered to catch up on all syntactic and semantic checking which has been deferred. Such a check will evaluate any given attribute only once, avoiding repeated evaluations of entire scopes. (Both of these features have been implemented.)

Another approach would be to automate the decision whether to check, by only doing checks when the user leaves an area with a certain granularity, such as a scope. As long as the decision can be made before the incremental parse begins (after the firewall has been determined), it is easy to defer the evaluation. Since avoiding the evaluation uses the same mechanism as handling a syntax error (the nodes of the firewall are linked into one large error production, and their parent, an error node, is made the root of the tree) if a decision cannot be made until later, the work done since creation of the firewall can be thrown away. A savings still results if only a small part of the work has been done.

The hardest part of the above method for automating the decision is determining when a scope has been left. A scope can be detected in a Babel tree because the hash table for the associated STBB is attached to a node that can be considered "the root of the scope". One approach is to define the scope of a node as the nearest hash table on the path from that node to the root. Then the scope of the focus is easy to determine. Care must be taken, however, to ensure that reparsing of a portion of the tree does not change the hash table used for that scope, as it might when reevaluating it. The table must be reused when possible. (Recall that all attributes above the firewall are reevaluated. By definition, the hash table in question will always be above the firewall.) A disadvantage to this approach is that executable statement changes, whose reevaluations are less expensive, would also not be checked until a scope is left.

Another approach is to detect that the evaluations are going to be expensive, and avoid them. It is very hard to determine the cost before the parser starts, since it has little information about the syntactic or semantic structure of the changed part of the program. It can be detected shortly into the parse, by keying on reductions that will cause certain Lisp functions (i.e., **addste**, **hash**, **empty**, and so on.) to eventually be evaluated. A table of which productions fit this category can be created when the tables are read from the LDL table file, by inspection of the evaluation functions for each attribute of each production.

A much better solution would be to find a way to evaluate only the parts of the tree that use identifiers whose symbol table entries have changed. This is a hard problem in the context of attribute grammars. Some problems with this approach, and some ideas toward a solution are discussed here.

The primary problem is that attribute grammars are based on the notion of building up the "meaning" of a program from nothing. They do not deal with issues such as what to do when something that already exists is changed or is deleted. Another problem is that attribute grammars are very local in nature. To communicate a piece of information (such as a symbol table) from one part of a program (i.e., a declaration) to another part (i.e., an executable statement) it must be propagated one production at a time through the tree.

This property of attribute grammars is, however, a very desirable property when the original intent of attribute grammars is recalled: to define the semantics of a language. Such a definition is very high level, and can probably be developed by a language implementor much faster than a definition that specifies what to do in case of a change or deletion. A good system should handle changes and deletions internally, without forcing provisions for them in attribute grammars. Babel currently meets this goal, at the expense of considerable runtime overhead.

A solution might begin with the assumption that each use of an identifier is kept linked together in two lists. (1) A local list, rooted at the declaration of the identifier, of all uses of *that declaration* in the scope of the declaration. This list is useful when a change or deletion is made. (2) A global list, rooted in a global table, of all uses of that identifier in the entire program. This list is useful when an addition is made. Keeping these lists up to date is not an easy job itself, since the deletion problem occurs here, too. The easiest place to update the lists might be in the tree editor, after the firewall has been generated, before the incremental parser is invoked. The availability of the old version of the tree is needed to determine what to delete from the lists.

When a change is made causing the **hash** function to be called, the old version of the table (which will be attached to the same node the new table is to be attached to, if proper care in reusing existing nodes has been taken, unless a drastic change to the tree structure has been made) and the new version must be compared, to create a list of additions, deletions, and changes. Determining what has changed is easy, since all changes must be localized in one continuous stretch of the symbol table. A comparison could be made starting from each end; when differences are found the boundaries of the changed section have been determined. Then, a list of identifiers contained in each version of the symbol table is made. Those appearing in only the old table have been deleted, those appearing in the new table have been added, those appearing in both have been changed. The nonterminal to which the hash table is attached is called the **root** of the scope.

Now, all uses of these variables must reevaluated. Uses of changed or deleted variables can be found in their attached list of uses. Uses of added variables can be found by searching the global list. (Use of the global list could force reevaluation of more nodes than necessary. A more involved scheme might only flag those uses that are within the scope of the declaration.) For each token node found, the expanded superior graph would be added to the graph M of attributes to reevaluate. (The simplification made in Babel that only characteristic subordinate graphs are need would no longer be valid.)

The remaining problem is that since we are avoiding the propagation of the symbol table throughout the scope, most of the symbol table references which had previously been propagated will refer to an old version of the symbol table, and thus be wrong. A solution to this might be to adopt the convention that rather than passing a pointer to the symbol table itself throughout the scope, a pointer to the root of the scope (as defined above) to which the symbol table is attached is passed. This node would only be changed when a drastic structural change to the scope is made, requiring complete reevaluation. Again, care must be taken to ensure that the root of the scope does not change unless it must, to avoid extra evaluations, and avoid symbol tables pointing into an invalid part of the tree.

5.14.2. Space

The memory requirements of Babel are not extreme. The program size would easily fit in 64K bytes on a 16 bit minicomputer. The data size, not counting buffers for paging the tree in from disk, is large in the current implementation. While it might be possible to squeeze it into the 64K address space of a minicomputer, it would be necessary to use software routine to simulate paging of the tree, using only a few buffers. Performance would be degraded significantly.

The current implementation allocates enough virtual memory to hold the entire tree, and loads pages in from the file as they are needed. For this reason, an operating system with virtual memory is all but essential in order to have reasonable performance. The current implementation runs on a VAX running Berkeley Unix.

The space requirements of the disk file are another matter. No effort has been made in this experimental implementation to keep their size down, and it has turned out that their size is not only excessive, but that the excess size is increasing the CPU time needed by the editor. Current Babel trees without semantics average about 30 times the size of the equivalent text file. If semantic checking is done, the factor jumps to 300.

These numbers should not be taken to imply that Babel trees have to be so large. There are a number of things that could be done to make them smaller, which are planned for a future version. Some of these steps are outlined here.

Changes can be broken down into two categories: (1) making nodes smaller, and (2) having fewer nodes. All nodes in the tree currently have eight overhead fields, each occupying four bytes. The fields are (1) the type of the node, (2) a word of flag bits, (3-7) the parent, first child, last child, left sibling, and right sibling of the node, and (8) a link field used to link the firewall together.

Some of these fields can be discarded or made smaller. The type can be a single byte. The flags are not especially important (used to flag errors and changes to the tree) and could certainly be stored in one byte, if not totally eliminated. The link field is not really necessary, since the firewall can be linked together using the sibling fields and normal tree structure.

The structural fields seem indispensable at first, but two of them (say, left sibling and last child) could be eliminated by replacing the routine to look up the field by a routine to search a list of siblings to find the requested node. There would be a space/time tradeoff here.

A further improvement might be made by observing that several siblings linked together all have the same parent. The parent field could be eliminated by storing only one copy of the parent in the (otherwise null) right sibling field of the rightmost sibling, or the left sibling field of the leftmost sibling. (This idea is similar to the use of threads of [Knuth 1973].) Either a flag bit could be set indicating that this is the rightmost or leftmost sibling, or the corresponding field of the parent could be checked to see if it points back to the child node.

Another possibility for improvement is to reduce the size of tree pointers. In the current implementation they are four bytes, and represent an offset in the tree file.

Another possibility would be to have a separate table of tree node offsets, so that a tree pointer could be an integer index into this table. A two byte index would probably be sufficient.

Another apparent improvement might be to keep the text of tokens in a string table, and to reference the string table instead of saving the text of the token in each token node. Since tokens are often short[17], this improvement would probably not result in any space savings. It might, however, be useful in conjunction with the global linking together of identifiers discussed above.

A second approach is to cut down on the number of nodes. Using an abstract tree instead of an LR parse tree would have a dramatic effect on the size of the tree. If a method for incremental parsing using abstract trees could be developed, this change alone would be well worthwhile, not only to make the tree smaller, but to cut down on the number of attributes that must be evaluated, and to make the tree more natural for the user to move the focus around in.

In a tree with no semantics, most of the nodes are either tokens or nonterminals. Other than using an abstract tree, there is not much that can be done to reduce the number of nodes.

In a typical Babel tree with semantics, however, a typical nonterminal with three attributes might have three attribute nodes and five characteristic subordinate graph nodes. These nodes, plus the need for symbol tables, account for the factor of ten increase in the size of the tree. Since the attribute nodes and the characteristic subordinate graph nodes contain two and one integer values, respectively, (compared to 8 values of overhead) the tree size is strongly influenced by the number of nodes. A different representation of the characteristic subordinate graph, placed in the nonterminal node, would cut the tree size by a factor of two. Attributes might all be grouped in one node, for further savings.

[17]A measurement of one C and one Pascal program indicates that the average length of a token is slightly under two characters.

(Attributes should not be placed in the nonterminal node, since their values change more often than the nonterminal to which they are attached. Characteristic subordinate graphs, however, are fixed at the time the nonterminal is created.)

In summary, if all these space saving measures were taken, the size of the tree could be reduced to about 15 times the size of the text file. This tree would contain more information than the text file, such as the parsed structure, and the results of semantic evaluations. If no semantic evaluation were done, the figure could be reduced again by another factor of two.

CHAPTER 6

Language Description Language

6.1. Language

LDL describes language features that are needed by the Babel system. It does not attempt to define all aspects of the input language. The kinds of information contained in an LDL description are:

(1) Lexical information. This includes the forms of tokens, comments, white space, and reserved words that are needed by a scanner.

(2) Syntax, represented by a context free grammar.

(3) Prettyprinting rules. These are given in the form of special left and right shift items in the grammar, and newline items showing suggested locations for newlines.

(4) Semantics. Attribute grammars are used as the basis of the description. No attempt is made to describe all of the semantics. Static semantics can be described in as little or as much detail as desired by the language implementor.

This section describes the elements of the LDL formalism. See Appendix A for some examples of LDL descriptions.

6.1.1. Lexical Information

Regular expressions are the basis of the lexical model. Any language whose tokens are all regular expressions, which can be distinguished by their syntax alone can be processed. Language features that require a symbol table lookup in the scanner, such as those found in Algol 68 and C, are not handled. FORTRAN 66 and FORTRAN 77 are not handled, since they do not have reserved words and have conventions for separation of tokens that cannot be recognized with regular expressions. A dialect of FORTRAN having reserved words, requiring blanks to separate words, prohibiting blanks in the middle of a word, and eliminating the special significance of columns 1, 6, 7, and 72, could be described in LDL. FORTRAN 82 is expected to use a more conventional form and to be handled by this scheme.

By convention, names in MixedCase represent tokens and other lexical information, and names in lowercase are nonterminals.

The notation used is very high level, not at all like the typical coding of a scanner for a compiler. Three pieces of information are given as input:

(1) A list of reserved words.

(2) A list of "constant tokens" (those that have only one textual representation) such as ":=", "<", "<=", "+", etc. This category includes most punctuation but not reserved words. (It is actually possible to include reserved words here. For reasons of readability a separate section for reserved words was created.)

(3) A list of regular expressions for those tokens that are not constant (identifier, comment, integer constant, real constant, string constant, white space. etc.) The regular expression notation used is the same as that of Lex.

These regular expression descriptions are passed to Lex without interpretation, allowing all the features of Lex to be used. Unfortunately, such descriptions are frequently hard to read. An improvement to LDL might implement a level of "syntactic sugar" providing the language implementor with a more readable regular expression notation, such as the one used in [Geigerich 1979]. Since the regular expressions are currently viewed as strings by LDL, an additional benefit would be additional syntactic checking from the LDL language itself.

Here is an example of the lexical portion of an LDL description.

```
%reswords
        Begin           "begin"
        End             "end"
%constant
        Becomes         ":="
        Lt              "<"
        Le              "<="
        Plus            "+"
        Semi            ";"
%tokens
        Id              "[a-zA-Z][a-zA-Z0-9_]*"
        Intconst        "[0-9]+"
        String          "\\"([^\"]|\\\\")*\\""
        Comment         "\"/*\"([^*]|\"*\"[^/])*\"*/\""
        Whitespace      "[ \t\n]*"
```

Some special interpretations of these rules are made. The token **Id** is special because reserved words must be **Id**'s. **Whitespace** will be ignored when found. It is instead reduced to two integers, the number of newlines and the number of blank spaces preceding the next token. These two integers become a property of the next token. **Comments** are also ignored by the parser but are attached to the following "real" token (along with any white space preceding the comment) by the scanner so that they are not lost. (Since there are endmarker tokens, there is always a "next token" to which to attach a comment.)

6.1.2. Syntax

The language syntax is described using the same notation as Yacc. For example:

```
%grammar
        goal:           Begin stmtlist End;
        stmtlist:       stmt;
        stmtlist:       stmtlist stmt;
        stmt:           Id Becomes expr Semi;
        expr:           prim;
        expr:           expr Plus prim;
        prim:           Id;
```

6.1.3. Prettyprinting

Prettyprinting tells the editor exactly how programs should be indented. The programmer is not forced to format his program this way. A prettyprinting utility could reformat a program to conform to the prettyprinting standards described here.

Indenting rules are embedded in the grammar and are of three types:

%nl	indicates that a newline should appear at this point.
%rshift	indicates that the indenting level should shift one level to the right.
%lshift	indicates that the indenting level should shift one level to the left.

For example:

```
%grammar
        goal:           Begin %nl %rshift stmtlist %lshift End %nl;
        stmtlist:       stmt;
        stmtlist:       stmtlist stmt;
        stmt:           Id Becomes expr Semi %nl;
        expr:           prim;
        expr:           expr Plus prim;
        prim:           Id;
```

The above grammar would suggest a program indented as follows:

```
begin
        a := 3;
        b := a+5;
end
```

The actual program might be typed in by the programmer in any format.

6.1.4. Semantics

The semantics are specified using an attribute grammar formalism [Knuth 1968]. It is possible to specify as little or as much of the static semantics of a language as the implementor desires. One extreme is to omit all semantics, which causes the editor to check only syntax. The other extreme would be to specify an entire translator in the attribute grammar. The translator could leave assembly code in the tree. It is our intention that for languages with a good deal of static semantics, the attribute grammar should specify enough of the semantics to detect static errors, but not actually to generate code. Going much further than this would lead to portability problems (assumptions about the object machine) that probably do not belong in the editor.

The attribute grammar must be noncircular. Since an attribute can only be evaluated when all attributes it depends on have been evaluated, there is no evaluation order possible if an attribute indirectly depends on itself. Since attributes are only evaluated in syntactically correct programs, there are no "one pass" problems requiring a grammar to belong to a more restrictive class.

Each symbol of the grammar has a set of zero or more attributes associated with it. Each attribute has a *name* which need be unique only for that symbol. The name is translated to a small integer by the LDL processor. For example, if a symbol has three attributes, they will be numbered 0, 1, and 2. Attributes are individually stored in the tree in *attribute nodes*, that are attached as children of the symbol to which they apply. Each attribute is a single-word quantity, representing either an integer or a pointer into the tree. Since tree pointers can indirectly reference complex structures, such as symbol tables, this capability is quite powerful.

Character strings cannot be stored directly in attributes, but a tree pointer to a token whose text is the character string can be stored, achieving the same effect. For purposes of semantic checking, the names of identifiers have been the only strings needed for the existing implementations. Lisp S-expressions cannot be stored directly in an attribute, since the attribute, stored in the tree on disk, has a lifetime longer than the S-expression, stored in primary memory. Unless the language implementor wishes to build and execute dynamic Lisp programs[1], there is no reason to store an S-expression in the tree, since the tree structure provides equivalent capabilities.

Several attribute evaluation functions may be attached to each production. Each function is viewed as a definition of one of the attributes of a symbol in the production. The attribute is defined as a function of other attributes of symbols in the production and other available information, such as the text of a token.

[1]This feature is not present in Babel, but could easily be added by giving the language implementor access from Lisp to the interpreter functions *eval* or *apply*.

In the LDL description, for each function, three pieces of information must be specified: which attributes are *used*, which attribute[2] is *set*, and the *evaluation function*.

Most of the attribute grammar literature also requires attributes to be classified as either *inherited*, based on information propagated from higher in the tree; or *synthesized*, requiring information from descendants of itself or its siblings. Such information is normally used as a clue to a one pass evaluator, telling in what order to evaluate the attributes. Since our evaluator is incremental, this information is used only for error checking in LDL.

The notation for the evaluation function is one area that is usually left up to the designer. This flexibility has, unfortunately, led to a complete lack of standardization of attribute grammar descriptions, causing a portability problem.

The notation used here is based on the Lisp language. Lisp was chosen because a small subset of the language is universal, providing a great deal of power for the language implementor with only modest work for the implementor of the Babel system.[3] For each attribute, the three pieces of information needed are specified in parenthesized lists, enclosed in square brackets.

An attribute of a symbol is designated by the notation "$i.attr" which refers to the *attr* attribute of the i^{th} symbol on the right hand side of the production ($i = 0$ refers to the left hand side).

Attributes are not declared; if one is set that does not exist, it is dynamically created. The set of possible attributes on a symbol is static, however, since it is not possible to dynamically create attribute names or numbers. Hence, it would be possible to have attribute declarations. They were omitted only for simplicity of implementation of Babel. Declarations are not necessary in LDL for the same reason they are not necessary in FOR-

[2]It is possible for an evaluation function to set more than one attribute within the production. While this will work in the current implementation, the function will be invoked to define each attribute, resulting in wasted evaluations. For this reason, it is recommended that each attribute have a separate evaluation function.

[3]The prototype Lisp interpreter, including symbol table management, was implemented in less than a week.

TRAN: the translator can determine the set of identifiers used and create a symbol table at translation time.

It is an error to use an attribute that has not been set. The LDL processor checks for this, with some extra work. The editor also trivially checks for uninitialized attributes, during evaluation when the value is accessed. The LDL processor builds a table of attributes, indicating whether they are inherited or synthesized, (based on sets information in the evaluation function rules). It checks each production that should be assigning a value to an attribute (based on whether the symbol is on the left or right hand side of the production and whether the attribute is inherited or synthesized) to ensure that a value is assigned. The evaluation time check is made when the value of an attribute is used. If no attribute node with the proper attribute number can be found attached to the symbol, the attribute is uninitialized. This check has proved quite useful in practice, since the most common error made in the implementation of Asple and Pascal was the omission of an evaluation rule.

For example, the following rules pass the symbol table down the tree and the type of the expression up the tree.

```
expr:   expr Plus prim
        [       (uses $1.type $3.type)
                (sets $0.type)
                (       (check "Type clash" (eq $1.type $3.type) $0.self)
                        (set $0.type $1.type)
                )
        ]
        [       (uses $0.stab)
                (sets $1.stab)
                (set $1.stab $0.stab)
        ]
        [       (uses $0.stab)
                (sets $3.stab)
                (set $3.stab $0.stab)
        ]
        ;
```

The special attribute "self" produces a tree pointer to the node itself. This attribute (number −1) is not actually stored in an attribute node, and never depends on other attributes. In effect, self is a reserved attribute name.

The following rule uses the symbol table that was passed down the tree to determine the type of the ID and pass it back up the tree. The variable "tmp" is a local variable.

```
prim:  Id
       [       (uses $0.stab)
               (sets $0.type)
               (prog   (tmp)
                       (setq tmp (lookup $0.stab $1.text))
                       (check "undeclared variable" (ne tmp nil) $1.self)
                       (set $0.type (getattr tmp 2))
               )
       ]
       ;
```

Check, **lookup**, and **getattr** are built-in symbol table functions. (See section 6.2 for the meanings of these functions.) Attribute 2 would, in this example, be the type of the variable. (The language implementor might choose to represent types as integers, or, more likely, as pointers into a symbol table.)

The second and third rules in the **expr** rule above illustrate a common case. The actions do nothing but copy attributes up or down the tree. Since this is so common, a shorthand notation is provided:

```
[cp $0.stab $ 1.stab $ 3.stab]
```

The LDL processor expands this into the Lisp code shown above.

6.1.5. Compiler Help

It would cut down significantly on the work that a compiler must do for every compilation if the compiler could start with the Babel tree representation instead of a traditional textual representation. This section discusses possible extensions to LDL to make this job easier.

Many compilers first scan and parse the program, building an abstract tree as the parse proceeds. It would be straightforward to traverse the Babel tree, which is an LR parse tree, to produce the desired abstract tree, thus avoiding the problems of scanning, parsing, and error recovery in the compiler.

A Babel implementation that checks semantics leaves much more information in the tree that would be useful to the compiler, such as the symbol table, types of expressions, and results of applying scope rules. There would be a performance improvement if the compiler did not have to recreate this information.

A *parse-tree-to-abstract-tree* grammar which told how to build abstract trees from corresponding parse trees would be useful here. While not needed directly by the editor, such a grammar could specify portably how an abstract tree should be formed, using pieces of the Babel tree to build the abstract tree. Babel semantic attributes could be included in this abstract tree, using rules in the parse-tree-to-abstract-tree grammar.

This notion is very similar to the *string-to-tree* grammar used in [Geigerich 1979]. A notation based on their graphic notation, or a suitable ASCII version, could be added easily to LDL. A standard subroutine to create an abstract tree using tables generated from this parse-tree-to-abstract-tree grammar could be provided, which would be used by a compiler.

One problem with this approach is that different compilers for the same language might need different abstract trees; that is, one could argue that the particular style of abstract tree is a property of the compiler, not the language. There is, however, a movement toward the standardization of abstract trees for some languages, see [Goos 1981] for an example. Such a convenient tool might encourage abstract tree standardization even more.

6.2. Lisp

This section describes the dialect of Lisp used for attribute evaluation functions. This dialect does not support all of pure Lisp[4], since parts of it were not needed. Some additional primitive functions have been added for manipulating attributes, symbol tables, and the tree.

[4] **Label and lambda are not implemented, nor is the language implementor given access to eval or apply.**

There is no garbage collector. While there is no reason a garbage collector could not be implemented, the amount of memory allocated by **cons** once the language tables are read in has turned out to be negligible. Since Babel is implemented in a virtual memory environment,[5] a significant improvement to performance was made by never freeing anything, so there is no penalty for failing to garbage collect. Most of the dynamically allocated memory in Babel is used by either tree buffers (the entire tree is kept in virtual memory for performance reasons) or language tables, neither of which is ever freed. Most of the reusable data structures are in local variables on the stack. Those few places where memory is allocated and freed did not justify the overhead of a first-fit search through all of virtual memory (touching every page) upon every allocation. A more intelligent memory allocator would prompt reconsideration of this tradeoff.

An evaluation function is a Lisp S-expression which is evaluated for its side effects, typically assigning a value to an attribute. Any value returned by the S-expression is ignored.

The user can statically define callable functions in a separate section of the LDL description called **%functions**. This allows the definition of functions with parameters, providing the advantages of modular and unrepeated code, plus the power of recursion, without implementing **lambda** for unnamed functions. Since all atom names are stored as indices into the atom table, a function body can be found for a call in constant time. This implementation does have static (unnestable) scoping for functions, which is different from other Lisps.

[5]A VAX 11/780 running Berkeley 4BSD Unix.

Functions are defined with the **defun** syntax, for example:

```
%functions
(defun checkaddste (namenode istab a1 a2)
        (prog (ste)
                (set ste (lookone istab (name namenode)))
                (check "Multiple declaration" (eq ste tnull) namenode)
                (addste (name namenode) istab a1 a2)
        )
)
```

This defines a function called "checkaddste" with parameters "namenode" (a tree node of a token, whose name is being added to a symbol table), "istab" (the input symbol table) "a1" and "a2" (two symbol table attribute (STA) values being used to initialize the symbol table entry.) The function adds the given name and STA's to the symbol table, after checking for a multiple declaration, and returns the new symbol table.

The section that follows describes the built in functions in LDL. This set of functions has been more than sufficient for the existing language implementations. It is easy, however, to add more functions as needed, in addition to any code needed to implement the function itself (if it is not already built into Babel in some way) there are only 4 lines of code needed to make a function accessible from Lisp.

6.2.1. Pure Lisp

atom, car, cdr, cond, cons, nil, null, quote and t

These functions behave exactly as they do in pure Lisp. Their inclusion, along with the function capability below, insures that the dialect of Lisp here is universal. **nil** and **t** are atoms representing true and false. **Cons** takes two arguments and creates a new Lisp cell with those halves. **Car** and **cdr** take one argument and return the first or second half of the indicated Lisp cell. **Atom** returns **t** if its argument is an atom, **null** returns **t** if its argument is **nil**. **Quote** returns its argument, without evaluating it. **Cond** is the Lisp conditional function, accepting one argument which is a list of pairs (condition value). The first pair with a condition which evaluates to **t** will return the corresponding value. If there are no matches, **nil** is returned.

6.2.2. Common Lisp Extensions

defun, prog, progn, set and **setq**

These functions are usually present in actual Lisp implementations, and are needed for practical programs. (**defun** funcname (parameters) value) is used in the **%functions** section to define a function. This permits the function (funcname arguments) to be called, binding the arguments to the parameters, and evaluating and returning the value. (set variable value) and (**setq** variable value) have the side effect of evaluating the value, and assigning the result to the variable. The variable can be an atom or an attribute of the form $i.name, as described previously. The form setq is used for atoms, set for attributes. Otherwise the two are identical. The forms (**prog** (localvars) e1 e2 ...) and (**progn** e1 e2 ...) provide local blocks and compound statements. If local variables are listed, they are created and bound to those names. The expressions are evaluated, and the value of the last expression is returned.

6.2.3. Comparisons

eq, ge, gt, le, lt, ne and **equal**

These functions take two arguments and return **t** or **nil** as the comparison function is true or false. **Eq** and **ne** require that the two values evaluate to the same atom. The others test for numeric inequality. **equal** can be used to test for numeric equality.

6.2.4. Arithmetic Functions

add1, divide, minus, plus, sub1 and **times**

These functions provide simple arithmetic capabilities. They take two integer arguments, perform the indicated operation, and return an integer result. **Add1** and **sub1** take one integer argument and return the number plus or minus one, respectively. This arithmetic facility is compatible with that in most other Lisp implementations.

6.2.5. Symbol Table Functions

These functions provide access to symbol table building blocks (STBBs) and symbol table entries (STE's).

(**addste** name oldste a1 a2 ...)

This function adds an entry with the given name to a symbol table. The attributes are assigned the values a1, a2, and so on. Oldste is the old symbol table, the new symbol table is returned. Oldste is not changed.

(**empty** gp)

The **empty** function creates and returns an empty symbol table, with the global pointer field set to gp.

(**findattr** symno attrno)

This function is the result of expanding $symno.attr. When evaluated, the appropriate attribute is looked for in the current production in the tree, and the value is returned. This function can also be the first argument to set, which will cause the given attribute to be placed in the tree, replacing any old value.

(**getattr** ste i)

The **getattr** function is used to retrieve an attribute of an STE. The ith STA of the entry ste is looked up and returned. There is no **setattr** function, since its use would encourage violation of the nondestructive sharing property. An STE's values are set at creation.

(**hash** stbb anchor)

The **hash** function produces a hash table for a given STBB. The hash table, which is built on top of the STBB, is returned, producing a new, faster version of the same STBB. The anchor is a node in the symbol table to which the hashed symbol table is attached directly.

(**lookone** stbb string)

The string is looked for in the STBB. If found, the STE found is returned. Otherwise, **tnull** is returned.

(**lookup** stbb string)

Lookup is like lookone, but if the search fails, the STBB referenced in the global pointer is tried, then its global pointer, and so on until the string is found or a global pointer with value **tnull** is found.

(**name** token)

For a tree pointer to a token, the character string which is the text of the token is returned.

(**stjoin** stbb1 stbb2)

This function joins two STBBs into one, nondestructively. The combined STBB is returned.

(**check** string condition node)

This function is used to check for semantic errors in the program. It can be thought of as an assertion. The condition is evaluated. If it is true, there is no error, otherwise there is an error. The text of the error is the string, and the error text is attached to the given node in the tree. The node is checked for the error message; if the condition is true and an error message is found, the error message is removed; if the condition is false and no error message is found, one is attached to the node. If an error has already been found in this evaluation, no additional error messages will be attached, to avoid the "cascading error messages" problem.

6.2.6. Tree Access Functions

These functions are needed by an implementation that wishes to create and use code created data structures. Since these permit arbitrary access to, and changes of the tree, nothing can be guaranteed about the nondestructive sharing property. The implementor should use great care not to violate this property.

tnull

This built in atom has the value of a null tree pointer.

type

The **type** function returns, as an integer, the type of the node which is its argument.

parent, firstchild, lastchild, leftsibling and **rightsibling**

These functions return the corresponding neighbor of their argument.

(**newnode** type size)

The newnode function creates a new node with the given type. The number of bytes needed for data is passed in the size argument. The node created is returned.

(**getfield** node fieldnumber)

(**setfield** node fieldnumber value)

These functions access and set the value of some integer field in the given node. The fields are numbered from zero.

(**insert** node parent lsib)

This function inserts the given node into the tree, as a child of the given parent node. It is arranged that the left sibling of the node will be lsib. To insert as the first child, lsib should be **tnull**.

(**prune** node)

This function removes the given node from the tree.

6.3. LDL Processor

The LDL processor **ldlproc** is a tool used to preprocess LDL descriptions of languages to produce tables that the other Babel tools can use. Ldlproc has been implemented in the style of Babel in that it uses a Babel tree file for input rather than text. This not only simplifies the implementation of ldlproc by eliminating the need to scan and parse, but allows the user to use the editor directly on the source file in tree form.

Ldlproc takes two arguments, the name of the tree file for input and the name of the table file where the results will be placed. The processor makes heavy use of existing UNIX tools.

The first pass of ldlproc walks the tree looking for names of tokens, nonterminals, and reserved words, saving them in internal tables. The uses/sets information in attribute rules are used to build an attribute symbol table, and to build the dependency graph D[p] for each production. Each attribute is classified as either inherited or synthesized, based on the side of the production in which it is set. Attributes which are set on both sides (i.e., are both inherited and synthesized) are flagged as errors, since this condition always indicates a logic error in the description. This first pass is needed because forward references are possible. A more involved implementation might keep a symbol table using attribute rules.

The second pass turns the list of reserved words, constant tokens, and regular expression tokens into an input file for **lex**. The grammar is turned into an input file for **yacc**. The number and lists of tokens, nonterminals, reserved words, and productions are written out to a "tables" file. (Rather than store the text of the productions, an integer representation is written to the file.) The attribute information (numbers and dependencies of attributes for each symbol and production) are output to the *semantics* C source file. The Lisp evaluation functions are output to the semantics file in an easy to reparse form, changing $*i.attr*$ to (**findattr** i attrno), outputting atoms found by atom number, and accumulating an atom table. The atom table is then output to the semantics file. Atoms have one of the types *reserved* (built in functions and constants such as **car**, **check**, and **nil**), *integer*, *string*, or *identifier*.

The **lex** and **yacc** processors are invoked on their respective input files. This leaves four C source files: the tables, semantics, lex output, and yacc output. They are then compiled with **writeout**, a handwritten, language independent C source file, to produce an executable file. This file, when run, writes out the tables to the desired output file in the format needed by Babel.

CHAPTER 7

Conclusion

7.1. Summary

This dissertation has shown that it is possible to build a language editor such as the Babel system described here. A text editor style user interface is shown to work. We claim it can be made to be to be practical, and to be simple to learn and to use.

The effort required to "re-source" to another language is quite small, if only syntax checking is required, and moderate if semantic checking is also desired. LDL descriptions for the syntax of several languages, including Ada [Ichbiah1980a], Rigel [Rowe1979a], Lisp, a subset of C [Kernighan1978a], and LDL itself have been created. In addition, LDL descriptions for Asple with full semantic checking, and for Pascal, with partial semantic checking, have been created. It has been shown how to implement special features of Pascal and Ada, and since the semantic notation is based on a universal subset of Lisp, the semantics of any language, even those without noncircular attribute grammars, can be handled.

The user interfaces of language editors can be viewed as a spectrum, with template editors such as [Feiler1980a] at one end, and text interface editors such as Babel at the other end. Other language editors fall somewhere in between. This dissertation shows that all the simplicity and ease of learning of traditional text editors can be combined with the language dependent checking and programmer assistance capabilities of tree editors.

A set of primitive functions is described, which makes it possible to build any text editor interface on top of a tree data structure. Underlying methods for incremental scanning, parsing, and semantic analysis are described.

A notation for the description of languages, LDL, is described. This notation is suitable for any language with static lexical and syntactic requirements, so long as the language can be described using regular expressions and reserved words as a lexical model, LALR(1) $\cap$ RL(1) context free grammars as a syntactic model, and noncircular attribute grammars as a semantic model. While LDL was designed for programming languages, other computer languages in this class can also be described. Provisions for syntax-directed prettyprinting are discussed, and extensions to support generation of abstract syntax trees to help automate the production of compilers are suggested.

While the cost to build and maintain the additional information required by a system such as Babel is higher than a traditional screen-oriented text editor, the potential benefit, in terms of programmer time, is higher too. A useful analogy can be made with line oriented and screen-oriented editors. The cost for screen editors is higher than line editors, because of the extra work to keep an up-to-date copy of the program displayed on the screen. This extra cost has not deterred the majority of programmers from switching from line editors to screen editors. It is our belief that as language editors become available, programmers will turn to them, in spite of the additional cost, because language editors can do more for the programmer while he remains in the editor.

7.2. Suggestions for Future Work

While it has been shown in this dissertation, and in many other papers, how to build a working language editor, there is much more work to be done. Many improvements are possible, to improve the performance of language editors, and to enlarge the class of languages handled. This section describes some possibilities for future work.

Some carefully controlled experiments should be conducted, comparing the effects of different programming environments on programmer productivity. For example, a four way comparison between a line oriented text editor, a screen-oriented text editor, a text interface language editor such as Babel, and a template interface language editor such as

PL/CS, could be made. In order to avoid biased participants who are used to one particular kind of editor, such an experiment would be best conducted in an introductory programming class in a university. Measurements such as amount of time spent in the editor, total time logged on, and grade in the class could be made and analyzed. A number of two way /comparisons could also be made.

Current incremental parsing technology does not allow for general purpose incremental parsing from abstract syntax trees. Some language editors that use abstract trees restrict the command set allowed, to make the parsing technique goal driven, since it is straightforward to generate an abstract tree during a goal driven parse, and attach it into the rest of the tree using some specialized command. Another possibility is to translate between the abstract and LR syntax trees, causing considerable overhead. A method for direct incremental parsing with abstract syntax trees, possibly involving a restricted class of abstract trees, would lead to a large performance improvement to Babel.

An algorithm to avoid reevaluation of the entire scope of a declaration, when that declaration is changed, would be a significant improvement. Some ideas for such an algorithm are discussed in section 5.14.

The system currently contains no formal interface between the front end (user interface) and back end (buffer management/checking) of an editor. An interface between the two should be developed, allowing different front ends (for familiar text editors such as vi, EMACS, and so on) to be plugged into different back ends (an ordinary text editor buffer manager, and a Babel back end). Careful attention must be given to the display module; it probably belongs in the back end, but the user interface needs a high degree of control over it.

While the restrictions on the syntax class of languages handled (LALR(1) $\cap$ RL(1)) is not a serious restriction, the lexical class leaves out a number of real programming languages. The unusual lexical styles of FORTRAN and BASIC, for example, cannot be handled. Languages with user defined reserved words, such as C and Algol 68, are not handled. Certain lexical issues, such as whether upper/lower case distinctions are significant in identifiers, are not handled in the current system. A clean model that handles such real languages would be a clear step forward. Lacking this, provisions for a small number of "special cases" which could be hand coded into the scanner, or provisions for inclusion of language dependent code for this purpose, would extend the domain of the language editor.

Preprocessors cannot be handled efficiently using current technology. A method for efficient, incremental preprocessing would expand the class of languages handled to include languages where nearly all programmers use the preprocessor, such as C. A model of preprocessors encompassing many of those in current languages (C, PL/I, Bliss, assemblers, FORTRAN preprocessors, etc.) would also be a significant improvement.

While the Babel system allows the user to format comments exactly as desired, it provides little automatic help in the case of prettyprinting comments. Some investigation into the structure, positioning, and formatting of comments would be useful in the construction of language independent prettyprinters.

A powerful method for automatically prettyprinting the program as it is typed in, in a language independent manner, would be useful. Since syntax errors are common as a program is being typed in, and programs are almost always incomplete, a method based only on lexical information, or on partial syntactic information, would be useful. For a certain class of languages and prettyprinting rules, it would be possible to count certain tokens as "right shift" tokens and certain other tokens as "left shift" tokens. For example, **begin** might be a "right shift" token. Such tokens could be counted without regard to the syntax of the program. A method for finding such languages, given the higher level, more powerful prettyprinting notation of LDL, and automatic generation of the lists of tokens might be found.

Incremental compilers based on Babel style trees, with subtrees marked if they are changed, would provide a real speedup in performance, possibly overcoming the extra cost of processing in the editor. Ordinary compilers, which could start with the Babel tree, would not need to scan, parse, handle syntax errors, or generate symbol tables, since these tasks are already done by the editor.

There are many specific improvements that can be made to the Babel system, and to the LDL language. Further tuning can reduce the CPU time and disk space needed, even without further theoretical results to improve the algorithms. A redesign of the lexical part of LDL would allow a more readable lexical syntax, and more checking by the editor. A language other than Lisp might be more efficient for interpretation of semantics.

Further research is needed in some of these areas to improve the methods used to implement language editing systems such as Babel. Now that an implementation is complete, the performance problems are clear. These problems can be taken into account in a subsequent implementation, producing a faster editor. It is our belief that a production quality editor can now be built, using the methods described here, with reasonable performance.

Bibliography

Alberga 1979.
C. N. Alberga, A. L. Brown, G. B. Leeman Jr., M. Mikelsons, and M. N. Wegman, "A Program Development Tool," Report RC 7859, IBM T. J. Watson Research Center, Yorktown Heights, New York 10598 (September 1979).

Arnold 1980.
K. Arnold, *Screen Updating and Cursor Movement Optimization: A Library Package*, Computer Science Division, University of California, Berkeley (1980).

Brosgol 1980.
B. M. Brosgol, J. M. Newcomer, D. A. Lamb, D. Levine, M. S. Van Deusen, and W. A. Wulf, "TCOL_{Ada}: Revised Report on An Intermediate Representation for the Preliminary Ada Language," Technical Report CMU-CS-80-105, Carnegie-Mellon University, Computer Science Department (February 1980).

Ciccarelli 1978.
E. C. Ciccarelli, "An Introduction to the EMACS Editor," AI Memo 447, MIT AI Lab (January 1978).

Cleaveland 1973.
J. Cleaveland and R. Uzgalis, *What every programmer should know about grammar*, Dept. Computer Science, Univ. of California, Los Angeles (1973).

DEC 1972.
DEC, *Text Editor and Corrector Program Programmer's Reference Manual*, Digital Equipment Corporation (1972).

Demers 1981.
A. Demers, T. Reps, and T. Teitelbaum, "Incremental Evaluation for Attribute Grammars with Application to Syntax-directed Editors," *Eighth ACM Symposium on Principles of Programming Languages*, pp. 105-116 (January 1981).

Donzeau-Gouge 1980.
V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, *Programming Environments based on Structured Editors: The MENTOR Experience*, IRIA Laboria (July 1980).

Feiler 1980.
P. H. Feiler and R Medina-Mora, *An Incremental Programming Environment*, Carnegie-Mellon University, Computer Science Department (December 1980).

Feldman 1978.
S. I. Feldman, *Make: A Program for Maintaining Computer Programs*, Bell Laboratories (August 1978).

Ganzinger 1980.
H. Ganzinger, *Private Communication* 1980.

Geigerich 1979.
R. Geigerich, "Introduction to the Compiler Generating System MUG2," TUM-INFO-7913, Technische Universitat Munchen (May 1979).

Ghezzi 1979.
C. Ghezzi and D. Mandrioli, "Incremental Parsing," *ACM Transactions on Programming Languages and Systems* 1(1) pp. 58-70 (July 1979).

Ghezzi 1980.
C. Ghezzi and D. Mandrioli, "Augmenting Parsers to Support Incrementality," *Journal of the ACM* 27(3) pp. 565-579 (July 1980).

Goos 1981.
G. Goos and W. A. Wulf, *Diana Reference Manual*, Institut Fuer Informatic II, Univeritaet Karlsruhe, and Computer Science Department, Carnegie-Mellon University (March 1981).

Horton 1980.
M. R. Horton and W. N. Joy, "Termcap(5) Manual Page," *Berkeley Unix Programmers Manual*, (December 1980).

Ichbiah 1980.
J. D. Ichbiah, B. Krieg-Brueckner,, B. A. Wichmann, H. F. Ledgard, J. C. Heilard, J. R. Abrial, J. G. P. Barnes, M. Woodger, O. Roubine, P. N. Hilfinger, and R. Firth, *Reference Manual for the Ada Programming Language*, Honeywell, Inc., and Cii-Honeywell Bull (July 1980).

Jazayeri 1974.
M. Jazayeri, "On Attribute Grammars and the Semantic Specification of Programming Languages," Report No. 1159, Case Western Reserve University (October 1974).

Johnson 1978.
S. C. Johnson, *YACC: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill (July 1978).

Joy 1980.
W. N. Joy and M. R. Horton, "An Introduction to Display Editing with Vi," *Berkeley Unix Programmers Manual*, (September 1980).

Kernighan 1978.
B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall (1978).

Kernighan 1979.
B. W. Kernighan and M. E. Lesk, *LEARN - Computer Aided Instruction on UNIX*, Bell Laboratories (January 1979).

Knuth 1968.
D. E. Knuth, "Semantics of context-free languages," *Math. Systems Theory* 2 pp. 127-145 (1968). See also vol 6, page 95, 1971

Knuth 1973.
D. E. Knuth, "Threaded Trees," pp. 319 in *The Art of Computer Programming, Volume I*, Addison-Wesley (1973).

Ledgard 1977.
H. F. Ledgard, "Production Systems: A notation for defining syntax and translation of programming languages," *IEEE Transactions on Software Engineering*, (April 1977).

Lesk 1979.
M. E. Lesk and E. Schmidt, *Lex: A Lexical Analyzer Generator*, Bell Laboratories, Murray Hill (January 1979).

Morris 1981.
J. M. Morris and M. D. Schwartz, "The Design of a Language-Directed Editor for Block-Structured Languages," *ACM Special Interest Group on Programming Languages* 16(6) (June 1981).

Ossanna 1976.
J. F. Ossanna, *Nroff/Troff User's Manual*, Bell Laboratories (October 1976).

Persch 1980.
G. Persch, G. Winterstein, M. Dausmann, S. Drossopoulou, and G. Goos, "AIDA Reference Manual," Technical Report Nr. 39/80,, Institut fuer Informatik II, Universitaet Karlsruhe (November 1980).

Raiha 1980.
K Raiha, "Bibliography on Attribute Grammars," *SIGPLAN Notices* 15(3) pp. 35-44 (March 1980).

Reid 1980.
B. K. Reid and J. H. Walker, *Scribe Introductory User's Manual,* Unilogic, Ltd., 605 Devonshire St., Pittsburgh, PA 15213 (1980).

Reps 1981a.
T. Reps, *Private Communication.* May 1981.

Reps 1981b.
T. Reps, "Optimal-time Incremental Semantic Analysis for Syntax-directed Editors," TR 81-453, Cornell University, Ithaca, N. Y. (March 1981).

Rowe 1979.
L. Rowe and K. Shoens, "Data Abstraction, Views and Updates in RIGEL," *Proceedings of 1979 ACM Special Interest Group on Management of Data Conference,* pp. 71-81 (May 1979).

Sandewall 1978.
E. Sandewall, "Programming in the Interactive Environment: The Lisp Experience," *Computing Surveys* 10(1) pp. 36-71 (March 1978).

Stallman 1978.
R. Stallman, "Structured Editing with a Lisp," *Computing Surveys* 10(4) pp. 505-507 (December 1978). Surveyers' Forum

Teitelbaum 1979a.
T. Teitelbaum, "The Cornell Program Synthesizer: A Microcomputer Implementation of PL/CS," TR 79-370, Cornell University (July 1979).

Teitelbaum 1979b.
T. Teitelbaum, "The Cornell Program Synthesizer: A Syntax-directed Programming Environment," *ACM SIGPLAN Notices* 14(10) p. 75 (October 1979).

Teitelman 1977.
W. Teitelman, "A Display Oriented Programmer's Assistant," CSL 77-3, Xerox Palo Alto Research Center (March 1977).

Teitelman 1978.
W. Teitelman, *Interlisp Reference Manual,* Xerox Corporation, Palo Alto Research Center (1978).

van Wijngaarden 1975.
A. van Wijngaarden, B. J. Mailloux, J. E. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker, "Revised Report on the algorithmic language Algol 68," *Acta Informatica* 5(1-3)(1975).

Wegner 1972.
P. Wegner, "The Vienna definition language," *Computing Surveys* 4(1) pp. 5-63 (March 1972).

Wilcox 1976.
T. R. Wilcox, A. M. Davis, and M. H. Tindall, "The Design and Implementation of a Table Driven, Interactive Diagnostic Programming System," *Communications of the ACM* 19(11) pp. 609-616 (November 1976).

Yonke 1975.
M. Yonke, "A Knowledgeable, Language-Independent System for Program Construction and Modification," Rep. ISI-USC-RR 77-42, University of Southern California, Information Sciences Institute (October 1975). Also University of Utah Ph.D. dissertation, University Microfilms no. 76-10,349

Appendix A

Example LDL Descriptions

```
/*
 * LDL Description of ASPLE
 *
 * ASPLE (A Simple Programming Language Example) is a
 * small language, based on Algol 68.  It includes
 * integers, booleans, pointers to any type (including
 * pointers), if statements, while statements, and
 * simple input and output.  For more details, see
 * "A Sampler of Formal Definitions", ACM Computing
 * Surveys, June 1976, page 191.
 */

/* list of reserved words */
%reswords
            Begin    "begin"
            End      "end"
            Int      "int"
            Bool     "bool"
            Ref      "ref"
            Input    "input"
            Output   "output"
            If       "if"
            Then     "then"
            Else     "else"
            Fi       "fi"
            While    "while"
            Do       "do"
            True     "true"
            False    "false"

/* constant tokens - those with only one form */
%constant
            Semi     ";"
            Comma    ","
            Becomes ":="
            Plus     "+"
            Times    "*"
            Lparen   "("
            Rparen   ")"
            Eq       "="
            Neq      "/="

/* tokens which are not constant, but have regular expressions */
%tokens
            /* Note: ASPLE has no comments, so PL/1 style comments are added */
            Comment          "(\"/*\"([^*]|\"*\"[^/])*\"*/\")|(#[^\n]*\n)"
            Whitespace       "[ \t\n]+"
            Id               "[a-zA-Z]+"
            Number           "[0-9]+"

%grammar

program:            Begin %nl %rshift decl_train %nl                      program
                    stm_train %lshift %nl End %nl
                    /*
                     * Pass the symbol table up from the decls,
                     * hashed, down to the statements
                     */
                    [        (uses $2.stab)
                             (sets $3.stab)
                             (set $3.stab (hash $2.stab $2.self))
                    ]
                    ;
```

```
/*
 * the declarations make a symbol table, which will be passed
 * to the statements to use for checking.
 */
```

decl_train

```
decl_train:     declaration
                [        (uses $1.stent)
                         (sets $0.stab)
                         (set $0.stab (stjoin $1.stent (empty tnull)))]
                |
                decl_train %nl declaration
                [        (uses $2.stent $1.stab)
                         (sets $0.stab)
                         (set $0.stab (stjoin $2.stent $1.stab))]
                ;
```

declaration

```
declaration:    mode idlist Semi
                /* copy the mode of "mode" over into idlist */
                [cp $1.pmode $2.inmode]
                [cp $1.nrefs $2.innrefs]
                /* copy symbol table up from idlist to parent */
                [cp $2.outtab $0.stent]
                ;
```

mode

```
/*
 * A mode has two attributes: a primative mode (pmode)
 * and the number of references (nrefs).
 */
mode:           Bool
                [        (uses)
                         (sets $0.pmode)
                         (set $0.pmode 0 /* BOOLEAN */)
                ]
                [        (uses)
                         (sets $0.nrefs)
                         (set $0.nrefs 1)
                ]
                |
                Int
                [        (uses)
                         (sets $0.pmode)
                         (set $0.pmode 1 /* INTEGER */)
                ]
                [        (uses)
                         (sets $0.nrefs)
                         (set $0.nrefs 1)
                ]
                |
                Ref mode
                [        (uses $2.pmode)
                         (sets $0.pmode)
                         (set $0.pmode $2.pmode)
                ]
                [        (uses $2.nrefs)
                         (sets $0.nrefs)
                         (set $0.nrefs (plus $2.nrefs 1))
                ]
                ;
```

idlist

```
/*
 * An idlist inherits a mode, passes that mode to each id in the list,
 * which makes an ste and passes the collection of ste's back up to
 * the parent of idlist.
 */
idlist:         Id
```

```
                [       (uses $0.inmode $0.innrefs)
                        (sets $0.outtab)
                        (set $0.outtab
                                (addste (name $1.self) (empty tnull) $0.inmode $0.innrefs)
                        )
                ]
                |
                Id Comma idlist
                [cp $0.innrefs $3.innrefs]
                [cp $0.inmode $3.inmode]
                [       (uses $0.innrefs $0.inmode $3.outtab)
                        (sets $0.outtab)
                        (set $0.outtab
                                (addste (name $1.self) $3.outtab $0.inmode $0.innrefs)
                        )
                ]
                ;

/*
 * The symbol table is passed down the tree to be used in statements.
 */
stm_train:      statement                                               stm_train
                [cp $0.stab $1.stab]
                |
                statement Semi %nl stm_train
                [cp $0.stab $1.stab $3.stab]
                ;

statement:      asgt_stm                                                statement
                [cp $0.stab $1.stab]
                |
                cond_stm
                [cp $0.stab $1.stab]
                |
                loop_stm
                [cp $0.stab $1.stab]
                |
                transput_stm
                [cp $0.stab $1.stab]
                ;

asgt_stm:       Id Becomes exp                                          asgt_stm
                /* Pass the stab through to the expression */
                [cp $0.stab $3.stab]
                /* pass down the # refs expected on lhs */
                [       (uses $0.stab)
                        (sets $3.drefs)
                        (prog (ste)
                                (set ste (lookup $0.stab (name $1.self)))
                                (set $3.drefs (sub1 (getattr ste 1 /* NREFS */)))
                        )
                ]
                /* check the id is declared and compatible with the expr */
                [       (uses $0.stab $3.primmode)
                        (sets $2.idste)
                        (prog (ste)
                                (set ste (lookup $0.stab (name $1.self)))
                                (set $2.idste ste)
                                (check "undeclared id" (ne ste tnull) $1.self)
                                (check "type clash"
                                        (eq (getattr ste 0 /* PMODE */)
                                                $3.primmode)
                                        $0.self)
                        )
                ]
                ;

/*
 * Check that the exp is boolean, and pass the stab through to the stmts.
 */
cond_stm:       If exp Then %nl %rshift stm_train %nl %lshift Fi          cond_stm
                [cp $0.stab $2.stab $4.stab]
                [         (uses $2.primmode)
                          (sets $2.drefs)
                          (progn
                                    (set $2.drefs 0)
                                    (check "boolean expected"
                                              (eq $2.primmode 0 /* BOOLEAN */)
                                              $2.self)
                          )
                ]
                |
                If exp Then %nl %rshift stm_train %nl %lshift
                Else %nl %rshift stm_train %nl %lshift Fi
                [cp $0.stab $2.stab $4.stab $6.stab]
                [         (uses $2.primmode)
                          (sets $2.drefs)
                          (progn
                                    (set $2.drefs 0)
                                    (check "boolean expected"
                                              (eq $2.primmode 0 /* BOOLEAN */)
                                              $2.self)
                          )
                ]
                ;

/* Loop is just like if. */
loop_stm:       While exp Do %nl %rshift stm_train %nl %lshift End        loop_stm
                [cp $0.stab $2.stab $4.stab]
                [         (uses $2.primmode)
                          (sets $2.drefs)
                          (progn
                                    (set $2.drefs 0)
                                    (check "boolean expected"
                                              (eq $2.primmode 0 /* BOOLEAN */)
                                              $2.self)
                          )
                ]
                ;

transput_stm:   Input Id                                                  transput_stm
                /* input: check that variable is declared */
                [         (uses $0.stab)
                          (sets $1.idste)
                          (prog (ste)
                                    (set ste (lookup $0.stab (name $2.self)))
                                    (set $1.idste ste)
                                    (check "undeclared variable" (ne ste) $2.self)
                          )
                ]
                |
                /* output: pass stab to expression to check */
                Output exp
                [cp $0.stab $2.stab]
                [         (uses)
                          (sets $2.drefs)
                          (set $2.drefs 0)
                ]
                ;

/*
 * Expressions: pass symbol table down, generate types at leaves,
 * pass types back up, checking for compatibility at operators.
 * drefs is passed down and is the number of references the context
 * wants there to be.  There must be at least that many in the
 * actual value, and zero if binary operations are done.
 */
exp:            factor                                                  exp
                /* Just copy stab down and mode back up */
                [cp $0.stab $1.stab]
                [cp $0.drefs $1.drefs]
                [cp $1.nrefs $0.nrefs]
                [cp $1.primmode $0.primmode]
                |
                exp Plus factor
                /* copy stab down */
                [cp $0.stab $1.stab $3.stab]
                [cp $0.drefs $1.drefs $3.drefs]
                /* check modes, the primmodes have to be the same */
                [       (uses $1.nrefs $3.nrefs $1.primmode $3.primmode)
                        (sets $0.primmode)
                        (progn
                                (check "type clash"
                                        (eq $1.primmode $3.primmode)
                                        $0.self)
                                (check "no ref addition" (eq $0.drefs 0) $0.self)
                                (set $0.primmode $1.primmode)
                        )
                ]
                [       (uses)
                        (sets $0.nrefs)
                        (set $0.nrefs 0)
                ]
                ;

/* Same idea as exp */
factor:         primary                                                 factor
                [cp $0.stab $1.stab]
                [cp $0.drefs $1.drefs]
                [cp $1.nrefs $0.nrefs]
                [cp $1.primmode $0.primmode]
                |
                factor Times primary
                [cp $0.stab $1.stab $3.stab]
                [cp $0.drefs $1.drefs $3.drefs]
                [       (uses $1.nrefs $3.nrefs $1.primmode $3.primmode)
                        (sets $0.primmode)
                        (progn
                                (check "type clash"
                                        (eq $1.primmode $3.primmode)
                                        $0.self)
                                (check "no ref multiplication" (eq $0.drefs 0) $0.self)
                                (set $0.primmode $1.primmode)
                        )
                ]
                [       (uses)
                        (sets $0.nrefs)
                        (set $0.nrefs 0)
                ]
                ;

primary:        Id                                                      primary
```

```
/* variable.  Look it up and pass the mode up */
[       (uses $0.stab $0.drefs)
        (sets $0.nrefs)
        (prog (ste)
                (set ste (lookup $0.stab (name $1.self)))
                (check "undeclared variable" (ne ste tnull) $1.self)
                (set $0.nrefs (getattr ste 1 /* NREFS */))
                (check "ref mismatch" (ge $0.nrefs $0.drefs) $1.self)
        )
]
[       (uses $0.stab $0.drefs)
        (sets $0.primmode)
        (prog (ste)
                (set ste (lookup $0.stab (name $1.self)))
                (check "undeclared variable" (ne ste tnull) $1.self)
                (set $0.primmode (getattr ste 0 /* PMODE */))
        )
]
|
constant
/* bool or int constant.  pass the mode up. */
[cp $0.stab $1.stab]
[cp $1.nrefs $0.nrefs]
[cp $1.primmode $0.primmode]
[       (uses $0.drefs)
        (sets $1.ok)
        (prog (nd)
                (set nd (eq $0.drefs 0))
                (check "no constant refs" nd $1.self)
                (set $1.ok nd)
        )
]
|
Lparen exp Rparen
/* parenthesissed expression - pass mode of subexp up */
[cp $0.stab $2.stab]
[cp $0.drefs $2.drefs]
[cp $2.nrefs $0.nrefs]
[cp $2.primmode $0.primmode]
|
Lparen compare Rparen
/* comparison - pass boolean up */
[cp $0.stab $2.stab]
[cp $0.drefs $2.drefs]
[       (uses)
        (sets $0.nrefs)
        (set $0.nrefs 0)
]
[       (uses)
        (sets $0.primmode)
        (set $0.primmode 0 /* BOOLEAN */)
]
;
```

```
/*
 * You can only compare integers.  Check this.
 * Since compares always return boolean, primary knows this and we
 * don't bother to pass anything up from here.
 */
compare:        exp compop exp
                [cp $0.stab $1.stab $3.stab]
                [       (uses)
                        (sets $1.drefs)
                        (set $1.drefs 0)
```

```
                    ]
                    [       (uses)
                            (sets $3.drefs)
                            (set $3.drefs 0)
                    ]
                    [       (uses $1.primmode $3.primmode)
                            (sets $2.ok)
                            (progn
                                    (check "lhs not integer"
                                            (eq $1.primmode 1 /* INT */) $1.self)
                                    (check "rhs not integer"
                                            (eq $3.primmode 1 /* INT */) $3.self)
                                    (set $2.ok 0)
                            )
                    ]
                    ;

/* Comparison operator - syntax only */
compop:             Eq | Neq ;                                               compop

/*
 * Constants are boolean or integer.  Pass the right mode up.
 */
constant:           bool_constant                                            constant
                    [       (uses)
                            (sets $0.nrefs)
                            (set $0.nrefs 0)
                    ]
                    [       (uses)
                            (sets $0.primmode)
                            (set $0.primmode 0 /* BOOLEAN */)
                    ]
                    |
                    int_constant
                    [       (uses)
                            (sets $0.nrefs)
                            (set $0.nrefs 0)
                    ]
                    [       (uses)
                            (sets $0.primmode)
                            (set $0.primmode 1 /* INTEGER */)
                    ]
                    ;

/*
 * Syntax only for these two.
 */
bool_constant:      True | False ;                                           bool_constant

int_constant:       Number ;                                                 int_constant
```

```
        /* LDL description of LDL */

%reswords
        Reswords        "%reswords"
        Constant        "%constant"
        Tokens          "%tokens"
        Grammar         "%grammar"
        Functions       "%functions"
        Cp              "cp"

%constant
        Colon           ":"
        Semi            ";"
        Lbracket        "["
        Rbracket        "]"
        Lparen          "("
        Rparen          ")"
        Dollar          "$"
        Dot             "."
        Or              "|"

%tokens
        Id              "[%a-zA-Z][a-zA-Z0-9_]*"
        Intconst        "[0-9]+"
        String          "\\"([^\"]|\\\\")*\\""
        Comment         "\"/*\"([^*]|\"*\"[^/])*\"*/""
        Whitespace      "[ \t\n]*"

%grammar
                                                                sections
sections:               /* empty */
                |       section sections ;
                                                                section
section:                ressect
                |       consect
                |       resect
                |       gramsect
                |       funsect ;
                                                                ressect
ressect:                Reswords restail ;
                                                                restail
restail:                /* empty */
                |       res restail ;
                                                                res
res:                    String
                |       Id String ;
                                                                consect
consect:                Constant contail ;
                                                                contail
contail:                /* empty */
                |       con contail ;
                                                                con
con:                    Id String ;
                                                                resect
resect:                 Tokens retail ;
                                                                retail
retail:                 /* empty */
                |       re retail ;
                                                                re
re:                     Id String ;
                                                                gramsect
gramsect:               Grammar gramtail ;
```

```
gramtail:        /* empty */                                          gramtail
           |     production gramtail ;

funsect:         Functions funtail ;                                  funsect

funtail:         /* empty */                                          funtail
           |     fundef funtail ;

fundef:          sexpr ;                                              fundef

production:      Id Colon rules Semi ;                                production

rules:           rhs semantics                                        rules
           |     rules Or rhs semantics ;

rhs:             /* empty */                                          rhs
           |     rhs Id ;

semantics:       /* empty */                                          semantics
           |     semantics semrule ;

           /* The following can be [cp src dest] or [uses sets code] */
semrule:         Lbracket sexpr sexpr sexpr Rbracket                  semrule
           |     Lbracket Cp sexpr sexprseq Rbracket ;

sexpr:           atom                                                 sexpr
           |     Lparen sexprseq Rparen
           |     Dollar Intconst Dot Id ;

sexprseq:        /* empty */                                          sexprseq
           |     sexpr sexprseq ;

atom:            Id                                                   atom
           |     Intconst
           |     String ;
```

```
          /* LDL description of the default dialect of Lisp */

%constant
          Rparen          ")"
          Lparen          "("
          Quote           "'"
          Dot             "."
%tokens
          Whitespace      "[ ,\t\n]*"
          Intconst        "[0-9]+"
          Fpnumb          "[0-9]+\.[0-9]+([eE][+-]?[0-9]+)?"
          String          "\\"([^\"]|\\\\")*\\""
          Comment         ";[^\n]*"
          Atom            "[^()'| \n\t]+"
%grammar
sexprseq:           /* empty */                                   sexprseq
          |         sexpr sexprseq ;
sexpr:              atom                                          sexpr
          |         Lparen sexprseq Rparen
          |         Lparen sexpr Dot sexpr Rparen
          |         Quote sexpr ;
atom:               Atom                                          atom
          |         Intconst
          |         Fpnumb
          |         String ;
```

```
/*
 * LDL description of Pascal, with partial semantics checking.
 *
 * This description builds a block structured symbol table
 * containing constants, types, and variables, and knows
 * the difference between the three (but not between two
 * types or two constants).  It checks that types and constants
 * are used in contexts where they are expected, and that
 * types, constants, and variables are declared.
 */
%reswords
          And           "and"
          Array         "array"
          Assert        "assert"
          Begin         "begin"
          Case          "case"
          Const         "const"
          Div           "div"
          Do            "do"
          To            "to"
          Else          "else"
          End           "end"
          Extern        "extern"
          File          "file"
          For           "for"
          Forward       "forward"
          Function      "function"
          Goto          "goto"
          If            "if"
          In            "in"
          Label         "label"
          Mod           "mod"
          Nil           "nil"
          Not           "not"
          Of            "of"
          Or            "or"
          Packed        "packed"
          Procedure     "procedure"
          Prog          "program"
          Record        "record"
          Repeat        "repeat"
          Set           "set"
          Then          "then"
          Downto        "downto"
          Type          "type"
          Until         "until"
          Var           "var"
          While         "while"
          With          "with"
          Oct           "oct"
          Hex           "hex"

%constant
          Dotdot        ".."
          Dot           "."
          Lparen        "("
          Rparen        ")"
          Semi          ";"
          Comma         ","
          Eq            "="
          Colon         ":"
          Plus          "+"
          Minus         "-"
          Arrow         "^"
```

```
        Lbracket        "["
        Rbracket        "]"
        Lt              "<"
        Gt              ">"
        Star            "*"
        Slash           "/"

%tokens
        Whitespace      "[ \t\b\r\f\n]+"
        Comment         "(\"(*\"([^*]|\"*\"[^)])*\"*)\")|(\"{\"[^}]*\"}\")|(#[^\n]*\n)"
        Id      "[a-zA-Z][a-zA-Z0-9]*"
        String  "'([^'\n]|'')*'"
        Int     "[0-9]+"
        Fpnumb  "[0-9]+\.[0-9]+([eE][+-]?[0-9]+)?"

%grammar
                                                              program
program:
        btab prog_hedr decls block Dot
                [cp $1.builtins $3.itab]
                [       (uses $3.stab)
                        (sets $4.stab)
                        (set $4.stab (hash $3.stab $3.self))
                ]
                [cp $4.stab $0.stab]
                |
        btab decls
                [cp $1.builtins $2.itab]
                [       (uses $2.stab)
                        (sets $0.stab)
                        (set $0.stab (hash $2.stab $2.self))
                ]
                ;
                                                              btab
btab:           btabchild
                /*
                 * We bury the built in symbol table over here to
                 * keep it from getting reevaluated.
                 */
                [       (uses)
                        (sets $1.builtins)
                        (prog (st)
                                (initialize)
                                (set st $1.builtins)
                                (cond ((eq st tnull)
                                        (set $1.builtins
                                                (empty (builtins $1.self))))
                                      (t nil))
                        )
                ]
                [cp $1.builtins $0.builtins]
                ;
                                                              btabchild
btabchild:      /* empty */ ;
                                                              prog_hedr
prog_hedr:
        Prog Id Lparen id_list Rparen Semi
                ;
                                                              block
block:
        Begin stat_list End
                [cp $0.stab $2.stab]
                /*
                 * Since this is only a partial implementation, we don't know
                 * about record fields.  This makes us cry wolf in "with" stmts
                 * because fields look like variables.  So we don't check
                 * any variables that are inside a with statement.
                 */
                [       (uses)
                        (sets $2.inwith)
                        (set $2.inwith 0)
                ]
                ;


/*
 * DECLARATION PART
 * input: itab (input symbol table)
 * output: stab (local symbol table)
 */
decls:
        decls decl
                [cp $0.itab $1.itab]
                [cp $1.stab $2.itab]
                [cp $2.stab $0.stab]
                |
        /* lambda */
                [cp $0.itab $0.stab]
                ;

decl:
        labels
                [cp $0.itab $1.itab]
                [cp $1.stab $0.stab]
                |
        const_decl
                [cp $0.itab $1.itab]
                [cp $1.stab $0.stab]
                |
        type_decl
                [cp $0.itab $1.itab]
                [cp $1.stab $0.stab]
                |
        var_decl
                [cp $0.itab $1.itab]
                [cp $1.stab $0.stab]
                |
        proc_decl
                [cp $0.itab $1.itab]
                [cp $1.stab $0.stab]
                ;

/*
 * LABEL PART
 */
labels:
        Label label_decl Semi
                [cp $0.itab $0.stab]        /* Don't bother checking labels */
                ;
label_decl:
        Int
                |
        label_decl Comma Int
                ;
```

```
/*
 * CONST PART
 */

const_decl:                                                        const_decl
            Const Id Eq const Semi
                    [cp $0.itab $4.stab]
                    [       (uses $0.itab $4.type)
                            (sets $0.stab)
                            (set $0.stab
                                    (checkaddste $2.self $0.itab 1 /*constant*/ $4.type))
                    ]
                    |
            const_decl Id Eq const Semi
                    [cp $0.itab $1.itab]
                    [cp $1.stab $4.stab]
                    [       (uses $1.stab $4.type)
                            (sets $0.stab)
                            (set $0.stab
                                    (checkaddste $2.self $1.stab 1 /*constant*/ $4.type))
                    ]
                    ;

/*
 * TYPE PART
 */

type_decl:                                                          type_decl
            Type Id Eq type Semi
                    [cp $0.itab $4.itab]
                    [       (uses $4.stab $4.type)
                            (sets $0.stab)
                            (set $0.stab
                                    (checkaddste $2.self $4.stab 2 /*type*/ $4.type))
                    ]
                    |
            type_decl Id Eq type Semi
                    [cp $0.itab $1.itab]
                    [cp $1.stab $4.itab]
                    [       (uses $4.stab $4.type)
                            (sets $0.stab)
                            (set $0.stab
                                    (checkaddste $2.self $4.stab 2 /*type*/ $4.type))
                    ]
                    ;

/*
 * VAR PART
 */

var_decl:                                                            var_decl
            Var vid_list Colon type Semi
                    [cp $0.itab $2.itab]
                    [cp $2.stab $0.stab]
                    [cp $4.type $2.type]
                    [cp $0.itab $4.itab]
                    |
            var_decl vid_list Colon type Semi
                    [cp $0.itab $1.itab]
                    [cp $1.stab $2.itab]
                    [cp $2.stab $0.stab]
                    [cp $4.type $2.type]
                    [cp $1.stab $4.itab]
                    ;
```

```
/*
 * vid_list for var decl.
 * input: itab (input symbol table), type (type of these ids)
 * output: stab (output symbol table).
 */
vid_list:                                                              vid_list
              Id
                        [         (uses $0.itab $0.type)
                                  (sets $0.stab)
                                  (set $0.stab (checkaddste
                                                $1.self $0.itab 3 /* variable*/ $0.type))
                        ]
                        |
              vid_list Comma Id
                        [cp $0.itab $1.itab]
                        [cp $0.type $1.type]
                        [         (uses $1.stab $0.type)
                                  (sets $0.stab)
                                  (set $0.stab (checkaddste
                                                $3.self $1.stab 3 /* variable*/ $0.type))
                        ]
                        ;

/*
 * PROCEDURE AND FUNCTION DECLARATION PART
 * input: itab. output: stab.
 * Passes a reference to itab with this phead added as gtab to decls.
 * Could easily avoid 1 pass property by passing down completed stab
 * instead of current one (pass up and down when completed).
 */

proc_decl:                                                             proc_decl
              phead Forward Semi
                        [cp $0.itab $0.stab]
                        [cp $0.itab $1.gtab]      /* Dummy, won't be used */
                        |
              phead Extern Semi
                        [cp $0.itab $0.stab]
                        [cp $0.itab $1.gtab]      /* Dummy, won't be used */
                        |
              phead decls block Semi
                        [cp $0.itab $1.gtab]
                        [cp $1.ogtab $0.stab]
                        [cp $1.stab $2.itab]
                        [         (uses $2.stab)
                                  (sets $3.stab)
                                  (set $3.stab (hash $2.stab $2.self))
                        ]
                        ;
phead:                                                                 phead
              Procedure Id params Semi
                        [         (uses $0.gtab $6.type)
                                  (sets $0.ogtab)
                                  (set $0.ogtab
                                          (checkaddste $2.self $0.gtab procedure void))
                        ]
                        [         (uses $0.ogtab)
                                  (sets $3.itab)
                                  (set $3.itab (empty $0.ogtab))
                        ]
                        [cp $3.stab $0.stab]
                        |
              Function Id params Colon type Semi
```

```
                [       (uses $5.stab $5.type)
                        (sets $0.ogtab)
                        (set $0.ogtab
                                (checkaddste $2.self $5.stab function $5.type))
                ]
                [       (uses $0.ogtab)
                        (sets $3.itab)
                        (set $3.itab (empty $0.ogtab))
                ]
                [cp $3.stab $0.stab]
                [cp $0.gtab $5.itab]
                ;
params:                                                         params
        Lparen param_list Rparen
                [cp $0.itab $2.itab]
                [cp $2.stab $0.stab]
                |
        /* lambda */
                [cp $0.itab $0.stab]
                ;

param_list:                                                     param_list
        param
                [cp $0.itab $1.itab]
                [cp $1.stab $0.stab]
                |
        param_list Semi param
                [cp $0.itab $1.itab]
                [cp $1.stab $3.itab]
                [cp $3.stab $0.stab]
                ;

/*
 * PARAMETERS
 */

param:                                                          param
        pid_list Colon type
                [cp $0.itab $3.itab]
                [cp $3.stab $1.itab]
                [cp $1.stab $0.stab]
                |
        Var pid_list Colon type
                [cp $0.itab $4.itab]
                [cp $4.stab $2.itab]
                [cp $2.stab $0.stab]
                |
        Function pid_list Colon type
                [cp $0.itab $4.itab]
                [cp $4.stab $2.itab]
                [cp $2.stab $0.stab]
                |
        Procedure pid_list
                [cp $0.itab $2.itab]
                [cp $2.stab $0.stab]
                ;

pid_list:                                                       pid_list
        Id
                [       (uses $0.itab)
                        (sets $0.stab)
                        (set $0.stab (checkaddste $1.self $0.itab parameter 0))
                ]
                |
        pid_list Comma Id
                [cp $0.itab $1.itab]
                [       (uses $1.stab)
                        (sets $0.stab)
                        (set $0.stab (checkaddste $3.self $1.stab parameter 0))
                ]
                ;

/*
 * CONSTANTS
 */

const:                                                                  const
        String
                [       (uses)
                        (sets $0.type)
                        (set $0.type array)
                ]
                |
        number
                [cp $1.type $0.type]
                [cp $0.stab $1.stab]
                |
        Plus number
                [cp $2.type $0.type]
                [cp $0.stab $2.stab]
                |
        Minus number
                [cp $2.type $0.type]
                [cp $0.stab $2.stab]
                ;
number:                                                                 number
        Id
                [       (uses $0.stab)
                        (sets $0.type)
                        (prog (ste)
                                (set ste (lookup $0.stab (name $1.self)))
                                (check "Undeclared constant" (ne ste tnull) $1.self)
                                (check "Constant expected"
                                        (equal (getattr ste 0) 1 /*constant*/) $1.self)
                                (set $0.type (getattr ste 1))
                        )
                ]
                |
        Int
                [       (uses)
                        (sets $0.type)
                        (set $0.type integer)
                ]
                |
        Fpnumb
                [       (uses)
                        (sets $0.type)
                        (set $0.type real)
                ]
                ;
const_list:                                                             const_list
        const
                [cp $0.stab $1.stab]
                |
        const_list Comma const
                [cp $0.stab $1.stab]
                [cp $0.stab $3.stab]
                ;
```

```
/*
 * TYPES
 */
                                                              type
type:
        simple_type
                [cp $0.itab $1.itab]
                [cp $1.stab $0.stab]
                [cp $1.type $0.type]
                |
        Arrow Id
                [cp $0.itab $0.stab]
                [       (uses)
                        (sets $0.type)
                        (set $0.type pointer)
                ]
                |
        struct_type
                [cp $0.itab $1.itab]
                [cp $1.stab $0.stab]
                [cp $1.type $0.type]
                |
        Packed struct_type
                [cp $0.itab $2.itab]
                [cp $2.stab $0.stab]
                [cp $2.type $0.type]
                ;
                                                              simple_type
simple_type:
        Id
                [cp $0.itab $0.stab]
                [       (uses $0.itab)
                        (sets $0.type)
                        (prog (ste)
                                (set ste (lookup $0.itab (name $1.self)))
                                (check "Undeclared type" (ne ste tnull) $1.self)
                                (check "Type expected"
                                        (equal (getattr ste 0) 2 /*type*/) $1.self)
                                (set $0.type (getattr ste 1))
                        )
                ]
                |
        Lparen eid_list Rparen
                /*
                 * Should enter eid_list in symbol table as enums.
                 * To do this we have to pass the symbol table down
                 * to all types and back up.  Later.
                 */
                [cp $0.itab $2.itab]
                [cp $2.stab $0.stab]
                [       (uses)
                        (sets $0.type)
                        (set $0.type enum)
                ]
                |
        const Dotdot const
                [cp $0.itab $0.stab]
                [cp $0.itab $1.stab]
                [cp $0.itab $3.stab]
                [cp $1.type $0.type]       /* Should do some checking */
                ;
                                                              struct_type
struct_type:
        Array Lbracket simple_type_list Rbracket Of type
                [cp $0.itab $6.itab]
                [cp $0.itab $3.itab]
                [cp $6.stab $0.stab]
                [       (uses)
                        (sets $0.type)
                        (set $0.type array)
                ]
                |
        File Of type
                [cp $0.itab $3.itab]
                [cp $3.stab $0.stab]
                [       (uses)
                        (sets $0.type)
                        (set $0.type file)
                ]
                |
        Set Of simple_type
                [cp $0.itab $3.itab]
                [cp $3.stab $0.stab]
                [       (uses)
                        (sets $0.type)
                        (set $0.type set)
                ]
                |
        Record field_list End
                [cp $0.itab $0.stab]
                [cp $0.itab $2.itab]
                [       (uses)
                        (sets $0.type)
                        (set $0.type record)
                ]
                ;
```

simple_type_list

```
simple_type_list:
        simple_type
                [cp $0.itab $1.itab]
                [cp $1.stab $0.stab]
                |
        simple_type_list Comma simple_type
                [cp $0.itab $1.itab]
                [cp $1.stab $3.itab]
                [cp $3.stab $0.stab]
                ;
```

eid_list

```
eid_list:
        Id
                [       (uses $0.itab)
                        (sets $0.stab)
                        (set $0.stab (checkaddste $1.self $0.itab 1 /*constant*/ enum))
                ]
                |
        eid_list Comma Id
                [cp $0.itab $1.itab]
                [       (uses $1.stab)
                        (sets $0.stab)
                        (set $0.stab (checkaddste $3.self $1.stab 1 /*constant*/ enum))
                ]
                ;

/*
 * RECORD TYPE
 */
```

field_list

```
field_list:
        fixed_part variant_part
                [cp $0.itab $2.itab]
                [cp $0.itab $1.itab]
```

```
                          ;
fixed_part:                                                       fixed_part
          field
                         [cp $0.itab $1.itab]
                         |
          fixed_part Semi field
                         [cp $0.itab $1.itab]
                         [cp $0.itab $3.itab]
                         ;
field:                                                            field
          /* lambda */
                         |
          id_list Colon type
                         [cp $0.itab $3.itab]
                         ;

variant_part:                                                     variant_part
          /* lambda */
                         |
          Case type_id Of variant_list
                         [cp $0.itab $4.itab]
                         |
          Case Id Colon type_id Of variant_list
                         [cp $0.itab $6.itab]
                         ;
variant_list:                                                     variant_list
          variant
                         [cp $0.itab $1.itab]
                         |
          variant_list Semi variant
                         [cp $0.itab $1.itab]
                         [cp $0.itab $3.itab]
                         ;
variant:                                                          variant
          /* lambda */
                         |
          const_list Colon Lparen field_list Rparen
                         [cp $0.itab $1.stab]
                         [cp $0.itab $4.itab]
                         ;

/*
 * STATEMENT LIST
 */

stat_list:                                                        stat_list
          stat
                         [cp $0.stab $1.stab]
                         [cp $0.inwith $1.inwith]
                         |
          stat_lsth stat
                         [cp $0.stab $1.stab]
                         [cp $0.stab $2.stab]
                         [cp $0.inwith $1.inwith]
                         [cp $0.inwith $2.inwith]
                         ;

stat_lsth:                                                        stat_lsth
          stat_list Semi
                         [cp $0.stab $1.stab]
                         [cp $0.inwith $1.inwith]
                         ;
```

```
/*
 * CASE STATEMENT LIST
 */

cstat_list:                                                      cstat_list
          cstat
                   [cp $0.stab $1.stab]
                   [cp $0.inwith $1.inwith]
                   |
          cstat_list Semi cstat
                   [cp $0.stab $1.stab]
                   [cp $0.inwith $1.inwith]
                   [cp $0.stab $3.stab]
                   [cp $0.inwith $3.inwith]
                   ;

cstat:                                                           cstat
          const_list Colon stat
                   [cp $0.stab $1.stab]
                   [cp $0.stab $3.stab]
                   [cp $0.inwith $3.inwith]
                   |
          /* lambda */
                   ;

/*
 * STATEMENT
 */

stat:                                                            stat
          /* lambda */
                   |
          Int Colon stat
                   [cp $0.stab $3.stab]
                   [cp $0.inwith $3.inwith]
                   |
          proc_id
                   |
          proc_id Lparen wexpr_list Rparen
                   [cp $0.stab $3.stab]
                   [cp $0.inwith $3.inwith]
                   |
          assign
                   [cp $0.stab $1.stab]
                   [cp $0.inwith $1.inwith]
                   |
          Begin stat_list End
                   [cp $0.stab $2.stab]
                   [cp $0.inwith $2.inwith]
                   |
          Case expr Of cstat_list End
                   [cp $0.stab $2.stab]
                   [cp $0.inwith $2.inwith]
                   [cp $0.stab $4.stab]
                   [cp $0.inwith $4.inwith]
                   |
          With var_list Do stat
                   [cp $0.stab $2.stab]
                   [cp $0.inwith $2.inwith]
                   [cp $0.stab $4.stab]
                   [
                            (uses)
                            (sets $4.inwith)
                            (set $4.inwith 1)
                   ]
```

```
                |
        While expr Do stat
                [cp $0.stab $2.stab]
                [cp $0.inwith $2.inwith]
                [cp $0.stab $4.stab]
                [cp $0.inwith $4.inwith]
                |
        Repeat stat_list Until expr
                [cp $0.stab $2.stab]
                [cp $0.inwith $2.inwith]
                [cp $0.stab $4.stab]
                [cp $0.inwith $4.inwith]
                |
        For assign To expr Do stat
                [cp $0.stab $2.stab]
                [cp $0.inwith $2.inwith]
                [cp $0.stab $4.stab]
                [cp $0.inwith $4.inwith]
                [cp $0.stab $6.stab]
                [cp $0.inwith $6.inwith]
                |
        For assign Downto expr Do stat
                [cp $0.stab $2.stab]
                [cp $0.inwith $2.inwith]
                [cp $0.stab $4.stab]
                [cp $0.inwith $4.inwith]
                [cp $0.stab $6.stab]
                [cp $0.inwith $6.inwith]
                |
        Goto Int
                |
        If expr Then stat
                [cp $0.stab $2.stab]
                [cp $0.inwith $2.inwith]
                [cp $0.stab $4.stab]
                [cp $0.inwith $4.inwith]
                |
        If expr Then stat Else stat
                [cp $0.stab $2.stab]
                [cp $0.inwith $2.inwith]
                [cp $0.stab $4.stab]
                [cp $0.inwith $4.inwith]
                [cp $0.stab $6.stab]
                [cp $0.inwith $6.inwith]
                |
        Assert Lparen expr Rparen
                [cp $0.stab $3.stab]
                [cp $0.inwith $3.inwith]
                ;
assign:                                                        assign
        variable Colon Eq expr
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                [cp $0.stab $4.stab]
                [cp $0.inwith $4.inwith]
                ;

/*
 * EXPRESSION
 */

expr:                                                          expr
        simple_expr relop simple_expr
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                [cp $0.stab $3.stab]
                [cp $0.inwith $3.inwith]
                |
        simple_expr
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                ;

simple_expr:
        simple_expr addop term
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                [cp $0.stab $3.stab]
                [cp $0.inwith $3.inwith]
                |
        Plus term
                [cp $0.stab $2.stab]
                [cp $0.inwith $2.inwith]
                |
        Minus term
                [cp $0.stab $2.stab]
                [cp $0.inwith $2.inwith]
                |
        term
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                ;

term:
        term divop factor
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                [cp $0.stab $3.stab]
                [cp $0.inwith $3.inwith]
                |
        factor
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                ;

factor:
        Not factor
                [cp $0.stab $2.stab]
                [cp $0.inwith $2.inwith]
                |
        Nil
                |
        String
                |
        Int
                |
        Fpnumb
                |
        variable
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                |
        func_id Lparen wexpr_list Rparen
                [cp $0.stab $3.stab]
                [cp $0.inwith $3.inwith]
                |
        Lparen expr Rparen
                [cp $0.stab $2.stab]
                [cp $0.inwith $2.inwith]
                |
        Lbracket element_list Rbracket
                [cp $0.stab $2.stab]
                [cp $0.inwith $2.inwith]
                |
        Lbracket Rbracket
                ;
```

element_list

```
element_list:
        element
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                |
        element_list Comma element
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                [cp $0.stab $3.stab]
                [cp $0.inwith $3.inwith]
                ;
```

element

```
element:
        expr
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                |
        expr Dotdot expr
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                [cp $0.stab $3.stab]
                [cp $0.inwith $3.inwith]
                ;

/*
 * QUALIFIED VARIABLES
 */
```

variable

```
variable:
        Id
                [       (uses $0.stab $0.inwith)
                        (sets $0.type)
                        (prog (ste)
                                (set ste (lookup $0.stab (name $1.self)))
                                (cond ((equal $0.inwith 0)
                                        (check "Undeclared variable"
                                                (ne ste tnull) $1.self))
                                )
                                (check "Type not allowed"
                                        (eq (equal (getattr ste 0) 2 /*type*/) nil)
                                        $1.self)
                                (set $0.type (getattr ste 1))
                        )
                ]
                |
        qual_var
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                [       (uses)
                        (sets $0.type)
                        (set $0.type integer)       /* guess */
                ]
                ;
```

qual_var

```
qual_var:
        array_id Lbracket expr_list Rbracket
                [cp $0.stab $3.stab]
                [cp $0.inwith $3.inwith]
                |
        qual_var Lbracket expr_list Rbracket
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                [cp $0.stab $3.stab]
                [cp $0.inwith $3.inwith]
                |
        record_id Dot field_id
                |
        qual_var Dot field_id
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                |
        ptr_id Arrow
                |
        qual_var Arrow
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                ;

/*
 * Expression with write widths
 */
wexpr:                                                    wexpr
        expr
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                |
        expr Colon expr
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                [cp $0.stab $3.stab]
                [cp $0.inwith $3.inwith]
                |
        expr Colon expr Colon expr
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                [cp $0.stab $3.stab]
                [cp $0.inwith $3.inwith]
                [cp $0.stab $5.stab]
                [cp $0.inwith $5.inwith]
                |
        expr octhex
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                |
        expr Colon expr octhex
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                [cp $0.stab $3.stab]
                [cp $0.inwith $3.inwith]
                ;
octhex:                                                   octhex
        Oct
                |
        Hex
                ;

expr_list:                                                expr_list
        expr
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                |
            expr_list Comma expr
                    [cp $0.stab $1.stab]
                    [cp $0.inwith $1.inwith]
                    [cp $0.stab $3.stab]
                    [cp $0.inwith $3.inwith]
                    ;
```

wexpr_list

```
wexpr_list:
            wexpr
                    [cp $0.stab $1.stab]
                    [cp $0.inwith $1.inwith]
                    |
            wexpr_list Comma wexpr
                    [cp $0.stab $1.stab]
                    [cp $0.inwith $1.inwith]
                    [cp $0.stab $3.stab]
                    [cp $0.inwith $3.inwith]
                    ;

/*
 * OPERATORS
 */
```

relop

```
relop:
            Eq
                    |
            Lt
                    |
            Gt
                    |
            Lt Gt
                    |
            Lt Eq
                    |
            Gt Eq
                    |
            In
                    ;
```

addop

```
addop:
            Plus
                    |
            Minus
                    |
            Or
                    ;
```

divop

```
divop:
            Star
                    |
            Slash
                    |
            Div
                    |
            Mod
                    |
            And
                    ;

/*
 * LISTS
 */
```

var_list

```
var_list:
            variable
                    [cp $0.stab $1.stab]
```

```
                [cp $0.inwith $1.inwith]
                |
        var_list Comma variable
                [cp $0.stab $1.stab]
                [cp $0.inwith $1.inwith]
                [cp $0.stab $3.stab]
                [cp $0.inwith $3.inwith]
                ;
```

id_list

```
id_list:
        Id
                |
        id_list Comma Id
                ;

/*
 * Identifier productions with semantic restrictions
 */
```

const_id *type_id* *array_id* *ptr_id* *record_id* *field_id* *proc_id* *func_id*

```
const_id:
        Id
                ;
type_id:
        Id
                ;
array_id:
        Id
                ;
ptr_id:
        Id
                ;
record_id:
        Id
                ;
field_id:
        Id
                ;
proc_id:
        Id
                ;
func_id:
        Id
                ;
%functions
/*
 * This avoids having to quote lots of stuff.  These are all essentially
 * manifest constants whose value doesn't matter, e.g. enumerated types.
 */
(defun initialize ()
    (progn
        (set abs (quote abs))
        (set arctan (quote arctan))
        (set boolean (quote boolean))
        (set char (quote char))
        (set chr (quote chr))
        (set constant (quote constant))
        (set cos (quote cos))
        (set enum (quote enum))
        (set eof (quote eof))
        (set eoln (quote eoln))
        (set exp (quote exp))
        (set false (quote false))
        (set file (quote file))
        (set function (quote function))
        (set input (quote input))
        (set integer (quote integer))
        (set ln (quote ln))
        (set maxint (quote maxint))
        (set odd (quote odd))
        (set ord (quote ord))
        (set output (quote output))
        (set pred (quote pred))
        (set real (quote real))
        (set round (quote round))
        (set sin (quote sin))
        (set sqr (quote sqr))
        (set sqrt (quote sqrt))
        (set succ (quote succ))
        (set text (quote text))
        (set true (quote true))
        (set trunc (quote trunc))
        (set type (quote type))
        (set variable (quote variable))
    )
)
(defun builtins (node)
    (hash
        (addste "false" (addste "true" (addste "maxint"
          (addste "integer" (addste "boolean" (addste "Boolean" (addste "real"
            (addste "char" (addste "text" (addste "input"
              (addste "output" (addste "abs" (addste "arctan"
                (addste "chr" (addste "cos" (addste "eof"
                  (addste "eoln" (addste "exp" (addste "ln"
                    (addste "odd" (addste "ord" (addste "pred"
                      (addste "round" (addste "sin" (addste "sqr"
                        (addste "sqrt" (addste "succ" (addste "trunc"
                          (addste "clock" (addste "wallclock"
                            (empty tnull)
                          function integer) function integer)
                        function integer) function enum) function real)
                      function real) function real) function integer)
                    function enum) function integer) function boolean)
                  function exp) function real) function boolean)
                function boolean) function real) function char)
              function real) function integer) 3 /*variable*/ file)
            3 /*variable*/ file) 2 /*type*/ file) 2 /*type*/ char)
          2 /*type*/ real) 2 /*type*/ boolean) 2 /*type*/ boolean) 2 /*type*/ integer)
        1 /*constant*/ integer) 1 /*constant*/ boolean) 1 /*constant*/ boolean)
    node)
)

/*
 * Routine like addste but checks first to make sure it isn't already
 * declared.  namenode is not the name, but the node containing the name,
 * so an error message can be placed in the tree if necessary.
 */
(defun checkaddste (namenode istab a1 a2)
        (prog (ste)
                (set ste (lookone istab (name namenode)))
                (check "Multiple declaration" (eq ste tnull) namenode)
                (addste (name namenode) istab a1 a2)
        )
)
```

```
        /* LDL description of TEXT, a transparent arbitrary text language */
%tokens
        Word            "[a-zA-Z0-9_]+"
        Whitespace      "[ \t\n]+"
        Punctuation     "[^ \t\na-zA-Z0-9_]+"
%grammar
text:
        |       text Word
        |       text Punctuation ;
```