5-1983

Development of syntax-directed editor for Pascal

Chintana Promphan

Rochester Institute of Technology
School of Computer Science and Technology

DEVELOPMENT OF SYNTAX-DIRECTED EDITOR FOR PASCAL

by

CHINTANA PROMPHAN

A thesis, submitted to
The Faculty of the School of Computer Science and technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by:_____
Professor Lawrence A. Coon

_____
Professor Warren R. Carithers

_____
Professor Peter G. Anderson

May, 1983

Title of Thesis:

DEVELOPMENT OF SYNTAX-DIRECTED FDITOR FOR PASCAL

I, Chintana Promphan, hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

(CHINTANA PROMPHAN)

Date: June 15, 1983

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Dr. Lawrence A. Coon, for giving me the main idea of this thesis and for the guidance and information he gave me throughout the course of it. Without his help, this thesis could not have been completed. I would also like to thank Mr. Warren R. Carithers for the time he spent correcting my writing and for being a member of my committee. Lastly, I wish to thank Dr. Peter G. Anderson for the time he spent as a member of my committee.

Abstract

A syntax-directed editor for Pascal is created by using an ALOE (A Language Oriented Editor) implementation tool running under the UNIX* Operating System. An ALOE supports the construction and manipulation of language structures while guaranteeing syntactic correctness. Instantiations of an ALOE for Pascal are generated from a grammatical description of a language. Trees are represented internally by ALOE as an abstract syntax trees that the user manipulates directly, and then maps this internal representation to a concrete representation for display to the user.

Key Words and Phrases: syntax-directed editor, language oriented editor, structure editor, template, constructive command

---

*UNIX is a Trademark of Bell Laboratories.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In the past, most program editors were based on the textual viewpoint. These editors executed commands which would interact with a program as a two dimensional page of characters. Such commands include moving the cursor around vertically and horizontally on the page, adding and deleting characters, lines, and regions of text.

A number of editors are now appearing which are based on the parse tree viewpoint. These editors execute commands which include moving the cursor around in a syntax tree, deleting subtrees, and instantiating templates for subtrees. The idea of syntax-directed editor is to superimpose a tree structure onto the text. Changes in the tree structure will cause the text parts to be correspondingly reordered. Syntax-directed editors help users create programs by preventing the entry of syntactically incorrect programs. They offer abbreviations for verbose constructs, and display programs in a pleasant, consistent fashion.

Language-oriented programming environments are designed around the language and thus can help and cooperate with the programmer in his task. In the process of creating, modifying and executing programs and systems, the programmer is

able to concentrate on his algorithms and on the structure of his systems, instead of on their specific textual representation.

Syntax-directed editors are language-oriented editors in which the programs and systems are created and modified according to the syntactic structure of the language instead of characters and lines. Some of the important characteristics of syntax-directed editors are the following [Medi82]:

A constructive approach. Programs and systems are built through constructive commands. Each command corresponds to a construct of the language. The burden of generating the concrete syntax is taken over by the editor.

Manipulation of programs and systems in terms of their structure. The programmer operates on his program or system in terms of its structure not its textual representation.

Abstract syntax trees. Programs are represented internally as syntax trees that are built by the editor from constructive commands created by the programmer. These trees are abstract syntax trees, whose nodes represent the language constructs. The elements of the concrete representation of languages, such as keywords, punctuation marks, separators, terminators, etc., are not kept in the tree.

Syntax tables. In syntax-directed editors generated from language descriptions, the syntactic knowledge of the languages obtained from these descriptions is kept by each editor in a collection of syntax tables. The editors use these tables to ensure the syntactic correctness of the programs being built.

Unparsing. The concrete representation of structures displayed in the screen is obtained by unparsing the internal abstract syntax tree representation

into a visual form. A set of rules, called unparsing schemes, specifies the visual representation of the construct.

Uniform interface for the environment. The user interface of a syntax-directed editor provides the means for communication between user and environment.

1.1. Goal of Thesis

The purpose of this thesis is to design the grammatical descriptions to generate a syntax-directed editor for Pascal [Jens74] by using an ALOE [Medi81b] (A Language Oriented Editor) implementation tool running under the UNIX* Operating System.

1.2. Disadvantages of Text-Oriented Editors

Richard Waters [Wate82] states that there seem to be three principal arguments which are used to support the idea that eliminating text oriented commands is either necessary or intrinsically beneficial.

(1) The textual viewpoint is obsolete. The structure oriented commands are much more convenient to use. As a result, nobody would want to use text oriented commands any more.

---

*UNIX is a Trademark of Bell Laboratories.

(2) The textual viewpoint is dangerous. It may be inconvenient to use structure oriented commands all of the time; however, this inconvenience is more than justified by the fact that this makes it impossible to create syntactically incorrect programs and therefore reduces errors.

(3) The textual viewpoint is incompatible with the structure oriented viewpoint. The implementation problems that would be created by having both structure oriented commands and text oriented commands are so severe that the text commands cannot be retained in a practical system.

1.3. Advantages of Text-Oriented Editors

Waters [Wate82] argues that each of these above arguments is fallacious.

(1) Structure oriented commands do many things better, but not everything. The fact that programs are displayed as two dimensional images composed of characters makes the textual viewpoint very natural. It is convenient to be able to enter simple expressions as strings of text rather than build them up hierarchically as trees.

(2) While a program is being modified by text oriented commands, it may temporarily pass through states where it

is not syntactically valid and therefore cannot be parsed. This can be dealt with by reparsing the program after groups of text editor commands and/or by extending the parsing algorithm so that it can tolerate certain kinds of syntactic errors.

(3) It is true that some syntactic errors become impossible if all program modifications are performed strictly from the point of view of the parse tree. However, this is only true for local properties. More global properties such as checking that every variable has a declaration, or that variables are used in ways that do not conflict with their declarations must be checked whenever the programmer changes the program.

1.4. Advantages of Syntax-Directed Editors

(1) Entry and modification of program text are guided by a grammar for the host programming language. The incorporation of the grammar into the editor guarantees syntactically correct programs; there is no need for syntactic error repair because such errors are prevented on entry [Teit81b].

(2) Editing commands can be applied to chunks of the program as defined by intermediate nodes of the parse tree. For example, the cursor can be moved around the

program with reference to the parse tree and subtrees can be copied and deleted [Wate82].

(3) The editor can check for various features of syntactic correctness as the programmer edits or enters the program. For example, it can check that the program is consistent with the programming language grammar and that each variable has a declaration [Wate82].

(4) The syntax-directed editor reduces typing effort by providing abbreviations for frequently occurring text elements such as keywords and by taking care of text formatting [Teit81b].

(5) A language-directed editor can form the basis of an integrated set of programming tools to support all phases of program development. The editor represents the syntactic structure of the program text in a parse tree, and this tree can subsequently be the subject of an interpreter, debugger, and/or code generator. Ultimately compilation as a separate and distinct step can be eliminated [Teit81b].

1.5. Disadvantages of Syntax-Directed Editors

(1) They are profligate consumers of computer resources, to the extent that they are usually designed for single user systems or multi-user systems carrying a low

interactive load [Morr81].

(2) Parsing usually consumes processing power, the parse tree devours storage, and there is no solution but to supply plenty of each [Morr81].

1.6. Previous Work

There are a number of program editors which operate on the basis of the parse tree viewpoint. These include the standard Interlisp editor [Teit78], Mentor [Donz75], The Cornell Program Synthesizer [Teit81a], Gandalf [Medi81a], PASES [Shap81], CAPS [Wilc76], and Barstow's display oriented Interlisp editor [Bars81]. Each of these editors more or less totally discards the textual viewpoint.

The Cornell Program Synthesizer is an interactive Programming environment with integrated facilities for creation, editing, executing, and debugging programs. The Synthesizer stimulates program conception at a high level of abstraction, promotes programming by stepwise refinement, and spares the user from the frustrations associated with syntactic detail.

The Synthesizer's editor is a hybrid between a tree editor and a text editor. Templates are generated by command, but expressions and assignment statements are typed

one character at a time. Errors in user-typed text are detected immediately because the parser is invoked by the editor on a phrase-by-phrase basis. By precluding the creation of syntactically incorrect programs, the Synthesizer lets the user focus on the intellectually challenging aspects of programming.

The language implemented for the Synthesizer was PL/CS [Conv76], an instructional dialect of PL/I [Teit76].

Creating a program in the Synthesizer is exactly analogous to deriving a sentence with respect to a context-free grammar for the given programming language. The following are the correspondences:

Placeholder	nonterminal symbol
template	right side of a production
command	the name of a production
insertion	derivation
file	derivation tree
cursor position	node of derivation tree
file display	sentential form

The Synthesizer is not only a syntax-directed editor, but also a programming environment that integrates interpretation and debugging facilities. The program cursor in the Synthesizer is a single character cursor as opposed to the area cursor of ALOE.

PASES is an interactive, residential programming environment that supports the creation of Pascal programs.

Its major components are a full screen structure editor and an interpreter, both operating on the same internal representation of Pascal structures.

The current design for the system includes the following main modules: parser, structure editor, text editor, display and file pretty-printer, semantic checker, interpreter, code generator, command processor, and a top-level control structure that coordinates these modules. The basic date structure in the system is a Pascal syntax tree. This is an abstracted parse tree generated by the parser. Trees of this type are stored in buffers. A buffer is essentially a pointer to the root of a syntax-tree, with an external name that can be referred by the user. The predefined buffers contain, among other things, templates for the most useful Pascal structures, including a program template, an equality expression template, etc. These buffers can be manipulated by the structure editor in the process of the top-down writing of a program.

The system derives almost all of its syntactic and semantic information about Pascal from an annotated BNF-like grammar. The parsing tables are generated by the UNIX program YACC, which uses this grammar as an input.

The MENTOR system [Donz80] is a structured editor for Pascal. MENTOR is based on parsing of input for all levels

of the language although it also supports some form of constructive editing. The MENTOR implementors decided to support parsing of input because users were more comfortable writing their programs as text as a result of their experience with text editors.

Users enter their programs as text but must deal with them structurally for editing and inspection. This might be confusing if the user enters his program as text, but edits it as a structure. MENTOR is not an integrated system: a program must be unparsed into a text file before being compiled separately.

MENTOR uses MENTOL, a tree manipulation language, as its command language. MENTOL is used to build routines to check context sensitive properties of Pascal programs and manipulate searches using pattern matching.

The Interlisp System is a very sophisticated programming system for LISP. Interlisp incorporates powerful facilities like structured editing, sophisticated debugging techniques, automatic error correction, the programmer's assistant and others. Input to Interlisp is given as text and it is parsed and inserted at the current position. Editing is done structurally as in MENTOR.

CHAPTER 2

PROJECT DESCRIPTION

Structure editing of a syntactically rich language like Pascal is new enough that actual experiments of a programming environment for Pascal like PASES [Shap81] were needed to understand its implications. Basic questions include how convenient is it to move with cursor commands on a tree structure rather than on a rectangular screen, or how natural is it to edit when the scope of the editing commands is the subtree of the node the cursor is pointing to rather than a letter or line.

2.1. The ALOE Implementation Tool

A Language Oriented Editor (ALOE) was first developed by the Department of Computer Science at Carnegie-Mellon University running under the UNIX Operating System. ALOE is a tool which supports the construction and manipulation of language structures while guaranteeing syntactic correctness. Instantiations of ALOEs are generated from a grammatical description of a language. Trees (i.e., programs) are represented internally by ALOEs as abstract syntax trees that the user manipulates directly. A simple language is provided for mapping this internal representation to a concrete representation for display to the user.

The ALOE generator produces a set of tables that define the language knowledge of each ALOE. These tables are compiled and then loaded with the rest of the editor to form a new ALOE. The ALOEs so generated are syntax directed editors. This means that construction and modification of programs or structures is done by following the abstract syntax of a language (which defines the structure of the language).

An ALOE can also be visualized as a constructive editor. Language constructs (such as variables, operators, expression, different kinds of statements) can be added, modified, or removed. The user communicates with ALOE in terms of these language constructs. The programmer constructs his program by inserting templates representing different language constructs and then filling the "holes" of those templates with other templates. Since ALOF knows which constructs are valid at any given point, it allows the programmer to insert a language construct only where it is syntactically correct. For example, instead of entering the character sequence for an "IF" statement in Pascal, the programmer calls on the template IF. The result is the insertion of this template:

```
if <expression> then
   <statement>
```

at the current program position, provided that this construct is syntactically correct in that context. Note that

this template contains the templates for expression and statement. The current program position is then advanced to <expression> so that it can be similarly expanded.

Internally ALOE represents the program as a "syntax tree". Each template corresponds to a node of a certain type in the tree. The holes of the template are the offspring of the node. They will be filled in with subtrees representing the expansion of those holes. Thus, the programmer actually constructs and manipulates a program tree without necessarily being aware of it. In order to make the tree readable by the user, ALOE uses an unparser that translates the syntax tree into a human-readable text. As part of its task, this unparser does the pretty-printing of the program. This unparsing process is driven by "unparsing schemes" specified in the grammatical description that describe the mapping from the abstract syntax to the concrete representation of each language construct.

The programmer interacts with ALOE through language commands (constructive commands) and editing commands. Language commands are used to insert new templates (e.g., the call for an IF template). Editing commands are used to manipulate the program tree (e.g., delete a subtree).

ALOE is extensible: Extended commands can be defined to implement other functions of an editing environment and to

incorporate language specific functionality. Additionally, action routines can be specified in the grammatical description of the language to be invoked by ALOE in certain situations. They allow the implementor of an ALOE to perform semantic checking, automatically generate program pieces, attach or review information stored in the tree, etc.

In order to create the ALOE for a language like Pascal, a description of the grammar for the language to be edited must be created. The generator is an ALOE instantiation (ALOEGEN) which can generate any syntax directed editor for any language according to its grammar. ALOEGEN is itself an interactive syntax-directed editor that is used to generate a grammatical description for a language.

2.2. Screen Organization

In every ALOE instantiation, the screen is divided into different windows. The main tree is displayed using a root window. Different contents of the tree can be displayed using different overlaying windows from a pool of tree windows. Clipped trees are displayed in clipped windows normally placed in the bottom half of the tree windows. Errors reported from action routines are displayed in an error window that normally overlays the tree windows.

Help information is displayed in a 'help' window normally placed at the right of the screen. The status window

is a one line window normally placed near the bottom of the screen and it is displayed only if the status mode is on.

2.3. The ALOE Generator

The ALOE generator produces an editing environment based on a grammatical description of a language. Every ALOE generated has three major components:

(1) The ALOE kernel common to all ALOEs. It understands and manipulates the internal representation of trees and is a language independent. It provides an extensive set of editing commands. It also provides the default implementation of the set of environment-specific functions.

(2) The syntactic tables produced from the grammatical description of the language. They provide ALOE's syntactic knowledge of the language, as well as the unparsing knowledge through the unparsing schemes.

(3) Implementation of action routines, extended commands and environment-specific routines. These provide the language specific behavior of an ALOE that is beyond its syntactic capabilities.

2.3.1. The grammar Description

The basic structure of a grammar has a ROOT node with a single offspring called GRAMMAR which contains five off-

strings as follows: language name, language root, terminal productions, nonterminal productions and classes.

2.3.1.1. Language Name

Each grammar has a language name associated with it. This is used to mark all tree files (programs) that are created and stored with an ALOE. The ALOE checks this name and does not attempt to edit a tree created by an ALOE for another language. The name of the file is the language name with the extension ".tr". For example, if the language name is 'PASCAL', the name of the file will be 'PASCAL.tr'. The language name is displayed on the screen as:

Language name: 'PASCAL'

2.3.1.2. Language Root

One of the non-terminals must be designated as the root of the grammar or start symbol. ALOEGEN automatically creates a non-terminal with the name of the root operator after the root operator is created. The root is displayed to the user as:

Language root: PROGRAM

2.3.1.3. Terminal Productions

The entire list of terminal operators is kept in a separate window. At the top level window, the terminal window is marked as:

>terminals<

While the cursor is positioned at this node, the window must be entered by the ._IN (<cursor-down>) command to expand the window into the terminal window. The user leaves the terminal window by getting to the root of the window and entering ._OUT (<cursor-up>). The terminal production is displayed on the screen as:

```
$opident  =
      $tertype ¦ action: $action ¦ synonym: $synonym ¦
      lex: $lex¦
      ($schexp) $scheme  ^
      $unparse;
$terminal
```

The character '¦' is used to separate the different parts of the production. The description of each terminal symbol consists of six parts as follows:

(1) Terminal Operator Name ($opident). The operator names should be alphanumeric and have no spaces. ALOEGEN

generates the table <language-name>.infop consisting of "#define"s for each operator in the language. Any non-alphanumeric characters (except space and tab) will be replaced by an underscore.

(2) Terminal Type ($tertype). There are several terminal types which are used to specify the type of the terminal operator, for example, variable type, real type, character type, etc. As the terminal type is entered, if the lexical routine is not yet filled in, a default is automatically supplied. The types and the defaults are listed in Appendix D.

(3) Action Routines ($action). These are used to expand on the syntactic capabilities of the basic AIOE, including semantic checking, automatic generation of program pieces, and communication with other parts of the editing environment such as access control, code generation, etc. If there is no action routine, then "<none>" is displayed. The action routines are useful in the integrated programming environment such as GANDALF.

(4) Synonym ($synonym). Synonyms resolve conflicts between variable names and constructive commands. For example, suppose the cursor is now positioned at the node "typeident", and the class of the constructive commands

at this point contains both "IDENT" for type identifier and 'INTEGER' for integer type. If the user wants this type to be integer and then types in "in", the ALOE will accept the word 'in' as an identifier instead of an integer type, which is wrong. If the INTEGER command has a synonym, such as ^I, the synonym may be used to unambiguously identify the type 'integer'. If there is no synonym for a terminal operator, then "<none>" is displayed.

(5) Lexical Routines. Lexical Routines are associated with the terminal operator so that simple lexical analysis of this operator can be performed according to the lexical routine given in the grammar. For example, the terminal operator 'IDENTIFIER' has a lexical variable so that every time the user enters an identifier name, ALOE will determine whether or not it is a legal identifier name. The default lexical routines associated with each terminal type are listed in Appendix D.

(6) Unparsing Schemes ($unparse). The definition of the unparsing schemes consists of a list of pairs. The first element of each pair ($schexp) is a list of numbers and ranges indicating which unparsing schemes are defined by the pair; the second element ($scheme) of the pair is a string that is the actual unparsing scheme, and has descriptions of the text, the

"syntactic sugar" required by the language (like the parentheses around the input list of a "read" statement) and the way the operator is to be formatted. Lists of unparsing schemes are in Appendix D. For example, the unparsing schemes of the comment operator are:

```
(0) "@>{ @c }"
(1) "@>(* @c *)"
```

The @> indicates that four spaces are to be inserted, and the @c indicates that a character string is to be inserted. So these schemes mean that when the unparsing scheme is zero the comment template is unparsed by inserting four spaces before the comment delimiter "{", and then followed by the content of the comment and "}" If the unparsing scheme is one, the comment delimiters will be '(*' and '*)' instead. Also, the first element of the pair can be in format (0:2,5) or (0,1,2,5) which means that the unparsing schemes 0,1,2, and 5 have the same format of actual unparsing scheme. The particular unparsing scheme may be selected with the .sc command, given as .sc <number> (.scheme <number>) command while editing, or by using the -u<number> command line switch to set the unparsing scheme used by ALOE to scheme<number> before starting an editing session. For example,

pcedit -u2 <filename>

calls the Pascal syntax-directed editor and selects an unparsing scheme two. Scheme zero is the default.

2.3.1.4. Non-terminal Productions

As with terminals, the list of non-terminals is kept in a separate window displayed as:

>nonterminal<

Non-terminal productions are displayed on the screen as:

```
$opident =
        $clspec ¦ action: $action ¦ synonym: $synonym ¦
        precedence: $precedence ¦ Non-filenode ¦
        $unparse;
$nonterminal
```

The description of each non-terminal symbol consists of seven parts as follows:

(1) Non-terminal Operator Name ($opident). Same as for terminal operator name.

(2) Offspring ($clspec). There are two types of non-terminals: fixed arity nodes (nodes with fixed numbers of offsprings) and variable arity nodes (nodes with

variable numbers of offsprings). Fix arity nodes are described by listing the set of offspring after the name of the operator. Each offspring is represented by the name of the class that specifies the legal operators for that field. Variable arity nodes are described by listing the name of the class from which elements of the list must be selected. This is displayed by enclosing the class name in angle brackets. Classes referred to as offspring of non-terminal operators are automatically generated if they do not already exist. Only the class identifier is supplied. The list of operator in the class must be filled in by the user.

(3) Action routines. Same as for the action routine of the terminal production.

(4) Synonym. Same as for the synonym of the terminal production.

(5) Precedence. Non-terminal operators optionally have precedence values associated with them. These are used to automatically parenthesize expressions while unparsing. The precedence values are only necessary for display purposes. Expressions, even though displayed in infix notation, are entered in prefix order with the operators serving as commands. For example, to enter

the expression (a+b)*c, one would enter its constituents in the following order: * + a b c. The parentheses are automatically displayed as required by the operator precedence. If no precedence is assigned, '<none>' is displayed.

(6) Filenode. This is used to determine whether or not the non-terminal has its subtree stored in a separate file (for storage and checkpointing purposes). The root operator is automatically made a filenode by the generator. The default is a non-filenode.

2.3.1.5. Classes

Classes are lists of types of offsprings, which define the set of terminal and nonterminal operators that are legal at that point. These are listed as the constructive commands when the novice mode is used. Classes are displayed in a separate window marked as:

>classes<

The class is displayed on the screen as:

```
$clident  =
            $opident;
$class
```

To enter to or exit from the class, press <cursor-down> and <cursor-up> respectively.

2.4. Building an ALOE for Pascal

STEP 1

Set the shell variable TERM to "vt100" and the environment variable EDITOR to "ex" so that ALOE will use ex as its text editor to edit a constant, long constant or text node. Add an environment variable GANDALFLIB which points to the directory which contains two files named libcmu.a and libgandalf.a. For example,

```
setenv GANDALFLIB /acct/misc/gandalf/lib
setenv EDITOR /usr/ucb/ex
set noglob; eval `tset -s -Q vt100`
```

STEP 2

Create the grammar for Pascal using the ALOE generator (ALOEGEN). This is in /acct/misc/gandalf/bin directory. The tree file the ALOEGEN saves will have the name given by the user to ALOEGEN with the extension '.tr'

added.

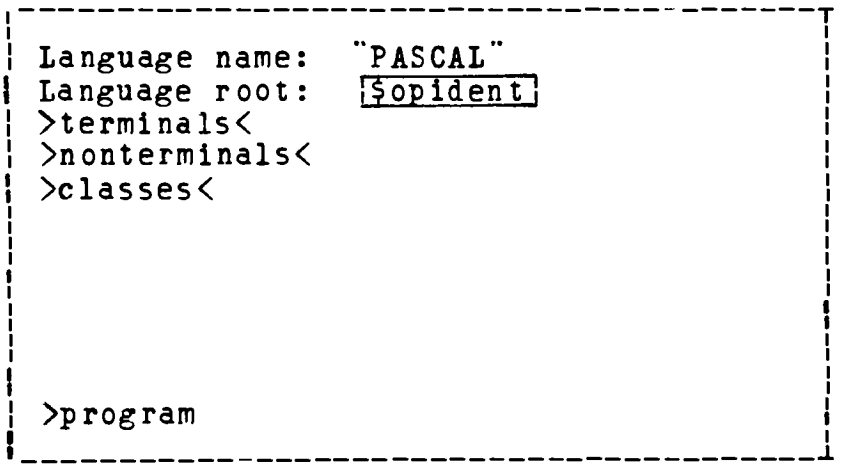
The following show the sequence of the ALOEGEN process for Pascal. After invoking "ALOEGEN", the system prompts for the program name which is a language name:

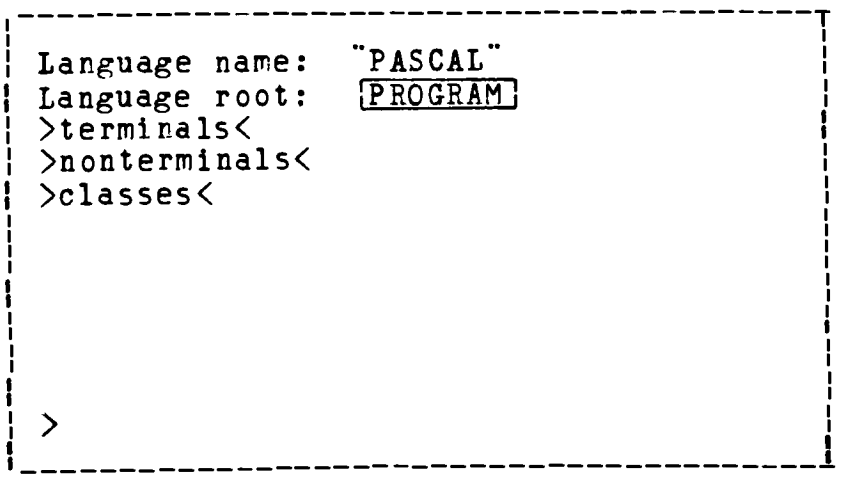
```
Program Name:
```

The implementor then types in the <language name>; for example, PASCAL The tree file named "PASCAL.tr" is created:

```
Language name:  "PASCAL"
Language root:  $opident
>terminals<
>nonterminals<
>classes<

>
```

The implementor then fills in all the unexpanded nodes: "$opident", "$terminals", "$nonterminals" and "$classes". The cursor is now positioned at the "$opident" which stands for the operator identifier. Typing in 'program' causes the "$opident" to be replaced.





Now, move the cursor to ">terminals<" (by using <cursor-right>) and enter the separate window that contains the terminal operators (by using <cursor-down>) and enter the terminal symbol descriptions (described in section 2.3.1.3). An example of the terminal operator window is:

```
/* terminal operators */

REAL   =
 {s} ¦ action: <none> ¦ synonym: <none> ¦ lex: <none> ¦
 (Ø)  "real";

REALCONST   =
 {r} ¦ action: <none> ¦ synonym: <none> ¦ lex: lexreal ¦
 (Ø)  "@c";

INTCONST    =
 {i} ¦ action: <none> ¦ synonym: <none> ¦ lex: lexinteger ¦
 (Ø)  "@c";
                                    .
                                    .
                                    .
                                    .
```

{s}, {r} and {i} are terminal node types ($tertype).

{s} stands for a static type that represents the concrete piece of the language, {r} for a real node that contains a real value and {i} for an integer node that contains an integer value. For more description of unparsing schemes, see Appendix D.

To leave the window, use <cursor-up>. Next, move the cursor to the non-terminal operator and repeat this process. Some of the example of the nonterminal operators of PASCAL are:

```
/* non-terminal operators */

PROGRAM =
     prog ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Filenode ¦
     (Ø) "@1";

PROG  =
     pgmhead block ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) '@n@1@2.';

PGMHEAD  =
     ident idents ¦ action: <none> ¦synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "program @1 (@2);";
                          .
                          .
                          .
                          .
```

The root operator "PROGRAM" contains one offspring "prog" that is a fixed arity node, where 'PROG' contains two offsprings of "pgmhead" and "block". If the offspring is a variable arity node, this will be displayed by enclosing the class name in angle brackets. @1 and @2 indicates the unparsing of the first and second offspring respectively, and @n indicates that the new line is to be inserted in the

output. Next are some of the PASCAL classes.

```
/* classes */

prog  =
      PROG COMPROG;

pgmhead =
      PGMHEAD ;

stat  =
      PROCCALLNOPAR READ READLN WRITE WRITELN REPEAT
      CASE WITH WHILE TFOR DWFOR GOTO IF IFELSE
      COMPOUNDSTAT EMPTY PROCCALL LABELSTAT REWRITE
      RESET ASSIGN COMSTAT STATCOM ;

exp   =
      IDENTIFIER NE PLUS MINUS MULT DIVIDE EQ MOD GT
      GE DIV LT LE NOT NEGATE AND OR IN INTCONST
      INDEXVAR FIELDESIGN POINTERVAR REALCONST SET EOF
      EOLN PROCCALL STRING NIL ;
                          .
                          .
                          .
                          .
```

On exit from ALOEGEN, two sets of tables will be generated. These are PASCALtbl.c (the table of data structures in C) and PASCAL.infop (a set of definitions which indicate the values assigned to the operators of the language). If the tables have been generated appropriately, the user will be asked whether or not the tables should be compiled. Before compiling these tables, the include statement in PASCALtbl.c has to be changed from

"usr/gandalf/include/datadefs.h" to "acct/misc/gandalf/include/datadefs.h" by editing this file. If there are classes which contain many operators, it might cause a 'line too long' problem when the implementor tries to edit this file. If this happens, the implementor has to shorten those lines by using another terminal and running through a program that will shorten all the lines that are too long in the file "PASCALtbl.c" before changing the 'include' statement. Then the implementor will be able to edit the "include" statement as mention before. Then the tables should be compiled, and the third table PASCALtbl.o will be generated.

STEP 3

- ALOE can be extended to have language or environment-specific behavior. These are three different kinds of mechanisms to provide these extensions: implementation of action routines, extended commands and environment specific routines.

Environment-Specific Routines

The ALOE system provides a variety of standard routines which can be replaced. When a particular ALOE is going to be generated, some or all of the routines can be replaced with environment-specific implementations. The object files for the standard (default) routines are collected in an

archive in '/acct/misc/gandalf/lib/DEFAULT.a'. If the user decides to modify some of the default routines, only the new routines must be written (using the same procedure names as the default routines) and compiled. The object files (.o) for these routines must be given to the loader before the DEFAULT.a archive file name. The loader will not pick routines from DEFAULT.a if they have already been defined. The environment specific-routine for a lexical variable shown in Appendix A was written to replace the default 'lexvariable' routine.

Action Routines

Action routines are specified as part of the grammatical description of the language. An action routine is called when an instance of the node associated with it is created, deleted, exited, etc. When an action call is made, two parameters are always passed: the node where the action is occurring and the type of the action call. The procedure specification for C procedures that are action routines should be:

```
actionROUTINE (thisnode, actkind)
struct tnode *thisnode;
int actkind;
{
   /* implementation code */
}
```

The values for the enumerated type represented by "actkind" are included in the ALOE library specification file "acct/misc/gandalf/include/ALOELIB.h". There are several types of action calls, for example, DELETE, INSERT, EDIT, NEST, TDELETE, UNNEST, RENAME and TRANSFORM. For more details of action, see [Medi81b].

For this thesis, some action routines such as checking for an empty character string, an identifier which is a reserved word, an undefined variable or type are included in the grammatical description.

Extended Commands

Extended commands are added to the basic set of editing commands to expand on the capabilities of an ALOE. Their user interface is the same as that of editing commands. The names and synonyms of extended commands should be chosen so that they do not conflict with those of the basic editing commands.

STEP4

Linking an ALOE. The command for linking an ALOE (ldaloe) is

```
ldaloe <language name> [<.o and .a files>]
```

for example,

```
ldaloe PASCAL lexvar.o aSTR.o aIDENT.o
```

where "PASCAL" is the language name, lexvar.o is an environment-specific routine, and aSTR.c and aIDENT.o are action routines. This will generate an ALOE in the file named <language name>.aloe. In this case the file name would be PASCAL.aloe.

2.5. Internal Representation of Syntax Tree

Each template corresponds to a node of a certain type in the tree. The unexpanded nodes, referred to as meta nodes, of the template are the offspring of that node. They will be filled in with subtrees representing their expansion. Figure 2.1 shows the display before the construction of an if statement in Pascal and the display resulting from its creation. Figure 2.2 shows the editing session for the construction of an 'if' statement. The program cursor is indicated by a rectangle in the figure. Note that the cursor is an area cursor that highlights the entire textual expansion of the current subtree. A single character cursor would be ambiguous in a structure context because the cursor would be placed at the first character of the first element of the list and it would not be clear whether it refers to that element or the entire list.

```
program add (input,output);
var  x,y;
begin
     [$stat]
end.
```

```
program add (input,output);
var  x,y;
begin
     if [$exp] then
        $stat;
     $stat
end.
```

Figure 2.1. Construction of an IF statement

Typed by user	Display	Syntax tree	Action
if	if |$exp| then $stat;	if / \\ <exp> <stat>	First, exp expanded to $exp = $exp.
=	if |$exp| = $exp then $stat;	if / \\ equal <stat>	The cursor is now positioned at the first $exp.
x	if x = |$exp| then $stat;	if / \\ equal <stat> / \\ x <exp>	By entering 'x', the cursor is advanced to the next $exp.
y	if x = y then |$stat|;	if / \\ equal <stat> / \\ x y	Enter 'y' for the second $exp; the cursor is then moved to a $stat template.

Figure 2.2. Editor session

CHAPTER 3

USER INTERFACE

An ALOE can be visualized as a constructive editor that understands the syntax of a language. Language constructs such as variables, operators, expression, statements, declarations, etc., can be added, modified and removed. The user communicates with ALOE in terms of these language constructs. The user constructs his program by inserting templates representing different language constructs and then filling in the unexpanded parts of those templates with other templates.

Internally, ALOE represents the program as a syntax tree. Each template corresponds to a node of a certain type in the tree. The unexpanded nodes of the template are the offspring of the node. These unexpanded nodes are referred to as meta nodes. They will be filled in with subtrees representing their expansion.

3.1. Editing Commands

Editing and constructive commands are specified using different naming conventions. Editing Commands are used to manipulate the program tree; they are prefixed with a period (".."). Constructive commands are used to insert new tem-

plates; they are prefixed with a comma (","). Only a sufficient number of letters to uniquely identify the choice must be given. There is no need to prefix the constructive command with ',' if there are no classes where the lexical routines for multiple terminal operators could be recognized simultaneously. An appropriate synonym can be used to more easily enter the terminals (nodes which represent leaves of the tree) without corresponding lexical routines. Command lines are terminated by a blank or <CR>. Several commands can be typed in a single line of input; ALOE will apply each command in sequence, and will update the display after the application of the last command. Drawbacks of this approach are especially evident when there are errors caused by a command, and the rest of the commands must be discarded or applied out of context.

A <CR> is interpreted as a command when applied at a meta node (unexpanded node); otherwise, it has no meaning. If the meta node is an element of a list, the <CR> indicates that the user has finished entering the list elements. At this point the meta node is deleted, and the cursor is moved to the next meta node. If the class of the current meta node contains an EMPTY operator, then the EMPTY operator is applied.

The implementation of cursor highlighting is optimal for the C-100 family of terminals and VT100 terminals. For

this thesis, an ALOE is to be generated to run on a GIGI terminal, so an inappropriate line update or cursor highlight might occur.

3.2. ALOE Modes

There are five mode settings in an ALOE:

(1) Cursor-follows mode indicates whether the cursor follows the tree structure or the current unparsing scheme. The default is to follow the tree structure. Note that this is an value mode not a boolean mode.

(2) Novice mode indicates that the help menu will always be displayed after every commmand. The default value is off.

(3) Edit mode indicates that the cursor will not be moved to the next meta node available in the tree after construction. The default value is off. This mode is only useful in ALOEs where different contexts are displayed using different overlaying windows.

(4) Status mode indicates whether the status window should be displayed. Its default value is off. It displays status information including the name of the current node, the class it belongs to, and the current values of the different modes.

(5) Window mode indicates whether the context-window window should be displayed. Its default value is 'off'.

3.3. Creating a Program

The editing process will be illustrated through the creation of a program to insert a blank at the beginning of each line of input resulting in a blank followed by a text for each line.

Next is the entire program that we are going to build.

```
    { insert leading blank }
program insertblk (input,output);
var     ch: char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.

>□
```

After logging into the Gandalf system, the user invokes the Pascal editor by executing the 'pcedit' command. The

ALOE will prompt for a program name; the user then types in a file named- for example, "insertblk". A file named 'insertblk.tr' will be created. The screen would look as follows:

The system is now ready to create a program. The rectangle represents the current cursor position, "<CR>" for the carriage return key and "^" for the control character. The non-terminal 'prog' appears on the screen as a placeholder for the program. Commands entered by the user will appear after the '>', and error messages will appear below that line. The novice user might want to see a menu after every command; this can be done by typing ".mode novice". After typing ".mode novice", the screen appears as follows:

```
[prog]                                                    PROG COMPROG


>.mode novice<CR>
 on or off [on]: <CR>
```

The constructive commands "PROG" and "COMPROG" appear on the screen. "PROG" is the command to start the program, and "COMPROG" is the command that calls the "comment" templates before beginning the program. All the constructive commands are listed in Appendix C.

Since one letter will make the choice unique, the user needs only to type in the first character of the constructive command- for example, 'p' for 'PROG' or 'c' for "COMPROG". After typing "c", the screen looks as follows:

```
[$comment] $prog                                          COMMENT


>
```

Since the available constructive command at this point contains only a "COMMENT" command, the user must type in "c" to insert the comment template. The system asks for the value of comment, the user then types in 'insert leading blank".

```
[$comment] $prog                                        COMMENT



>c
value: insert leading blank<CR>
```

The comment line appears on the screen, and ALOE prompts for the next comment:

```
    { insert leading blank }                            COMMENT
[$comment] $prog



>□
```

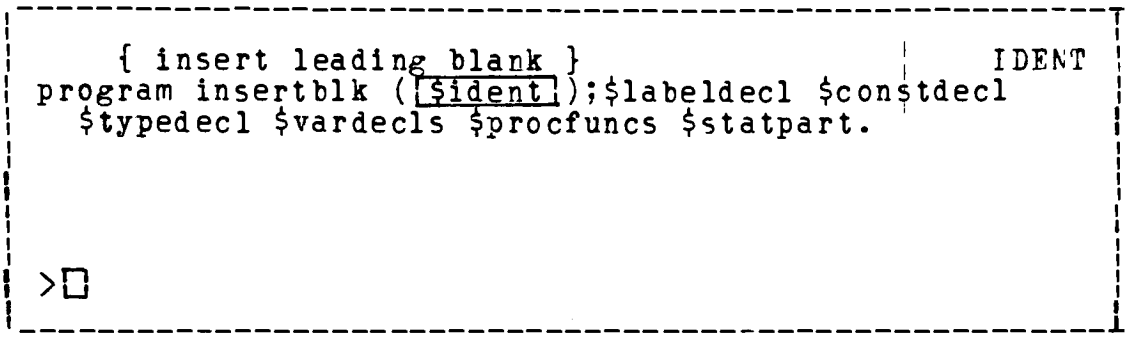
The system automatically generates the "{" and "}" delimiters. If the user decides to have '(*' and '*)' as

delimeters instead, type in a command '.sc 1' (.scheme 1) to change the unparsing to one. ".sc Ø" will change them back. The 'COMMENT' template at this point is designed as a list of comments so that the cursor keeps advancing to the next comment as long as the user continues entering lines of text. When the user is finished entering comments indicated by simply hitting <CR>, the cursor skips to "$prog". Again, the commands are the the same as before. Typing 'p' calls the template of Pascal skeleton:

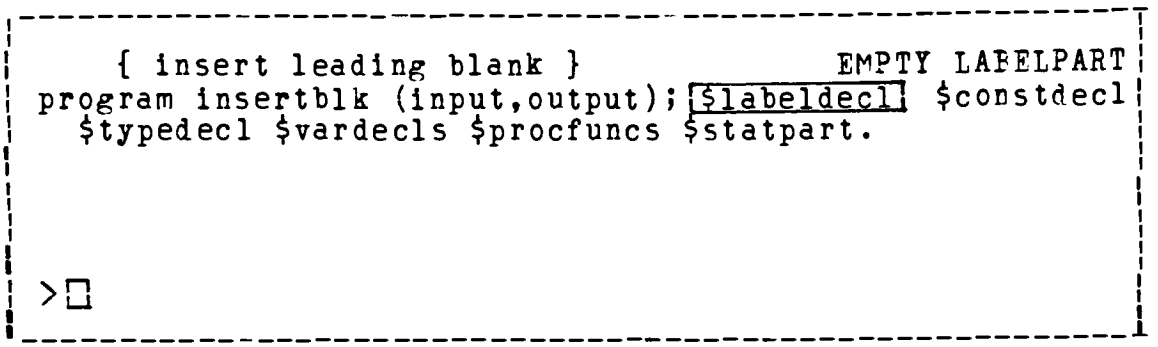
```
    { insert leading blank }                              IDENT
program [$ident] ($ident);$labeldecl $constdecl
  $typedecl $vardecls $procfuncs $statpart.



>
```

The meta node "prog" is replaced by the complete skeleton of a Pascal program. These eight new meta nodes identify locations where additional program elements can be inserted. The meta nodes are descriptive names that serve as visual cues. Any one of the eight nodes could be expanded next. The cursor skips over "program" because it is part of the generated program template and thus cannot be altered, and is positioned at "$ident", indicating that the program iden-

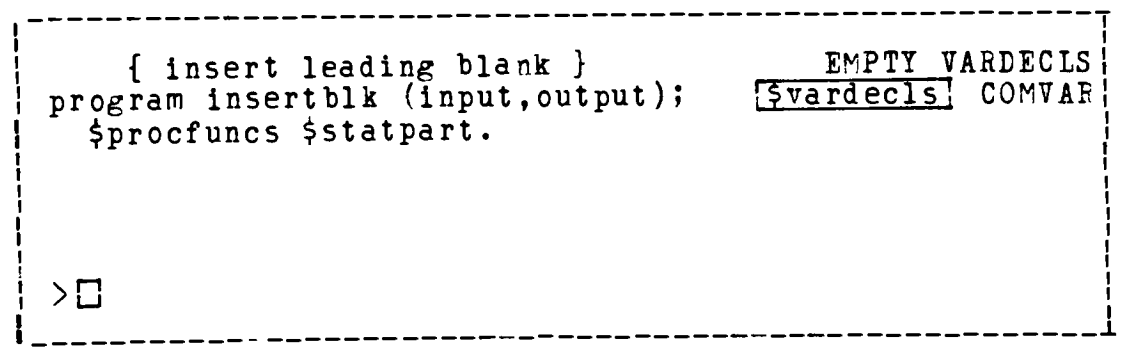
tifier should be entered. Typing the program name "insertblk" followed by <CR> replaces the "$ident" meta node:



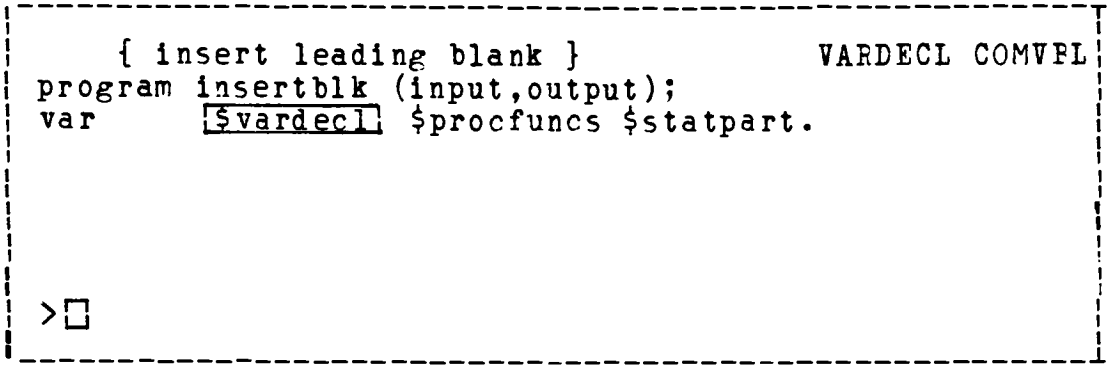
Next,the cursor is positioned at the list of file identifiers ($ident). After entering a file identifier, ALOE automatically generates a comma and prompts for more identifiers. Several commands can be typed in a single line of input separated by a blank. For example, instead of typing 'input<CR> output<CR>', the user might want to type the input line "input output<CR>"; ALOE will apply each item one at a time. When there are no more file identifiers to be entered, The user should simply hit <CR>. The "$ident" disappears and the cursor moves to "$labeldecl":



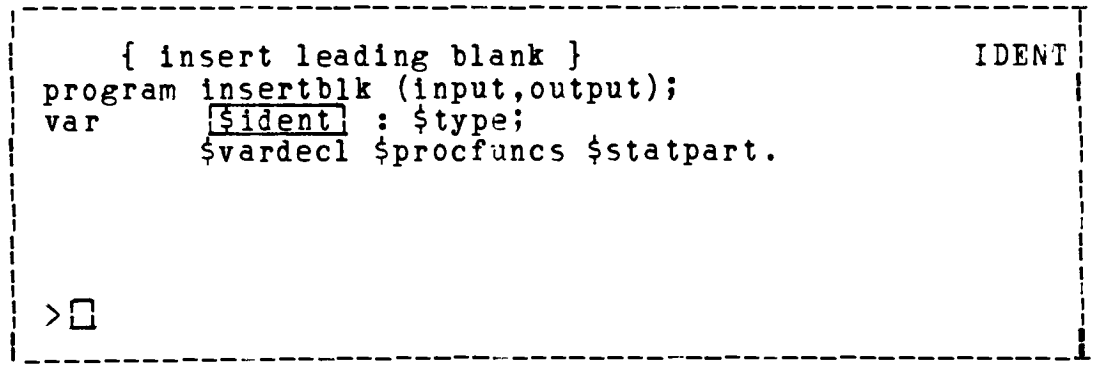
Since the command list contains "EMPTY", <CR> by itself could be applied to this operator. For this example, we do not need to declare labels, constants and types so that three repeated uses of <CR> will advance the cursor to "$vardecl":



Again, the constructive command contains 'EMPTY' for no variable declaration, "VARDECLS" for list of variable declarations, and 'COMVAR' for the comment and list of variable declarations. Typing "v" to call the variable declaration template will change the screen as follows:



Continuing to develop the sample program, we generate a variable definition template for variables by typing the command 'v' followed by '<CR>':



Typing "ch<CR>" replaces the "$ident" node:

```
    { insert leading blank }                                IDENT
program insertblk (input,output);
var     ch,[$ident] : $type;
        $vardecl $procfuncs $statpart.



>□
```

Again, the ALOE prompts for more identifiers. When we are done entering the identifiers, we type "<CR>". The cursor is positioned at '$type':

```
    { insert leading blank }            TYPEIDENT
program insertblk (input,output);       SCALAR({)
var     ch : [$type];                   SUBRANGE(})
        $vardecl $procfuncs $statpart.  PACKSTRUCT
                                        ARRAYTYPE([)
                                        RECORDTYPE(())
                                        SETTYPE FILETYPE
                                        POINTERTYPE(^)
                                        BOOLEAN REAL
                                        INTEGER CHAR
                                        TEXT


>□
```

A string appearing in parenthesis is a synonym for the operator it is next to. Synonyms are defined as alternate names by which commands can be invoked. If there are

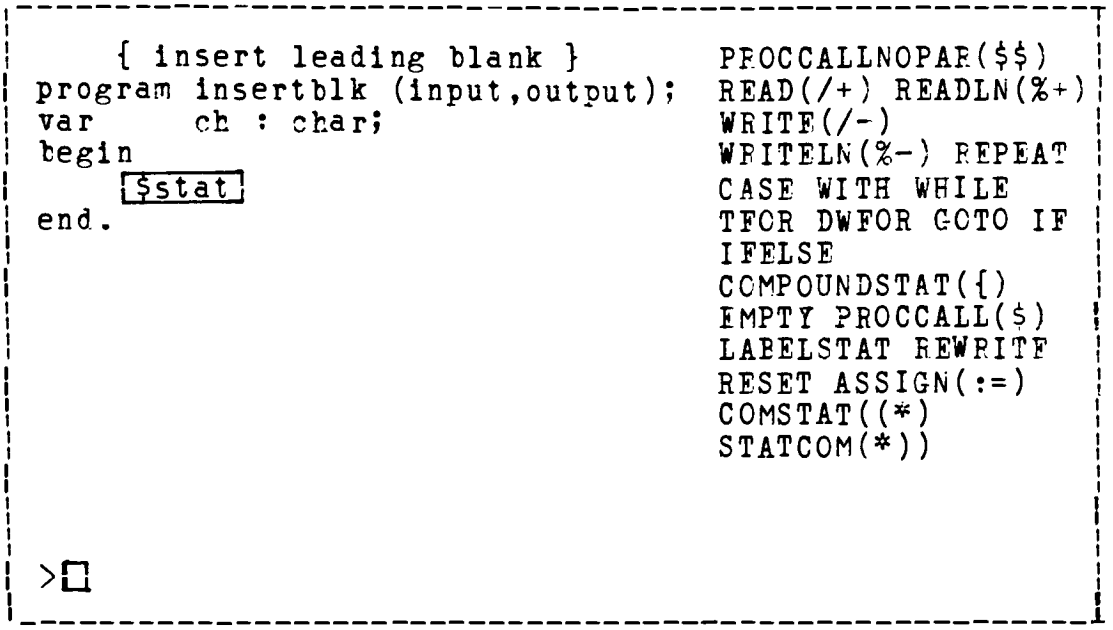
duplicate synonyms for operators in the same class, only the first one in the class will be matched. Editing command synonyms can be found by executing the .? command.

Since the command "CHAR" is designed to call the keyword 'char', typing ',c' followed by '<CR>' gives:

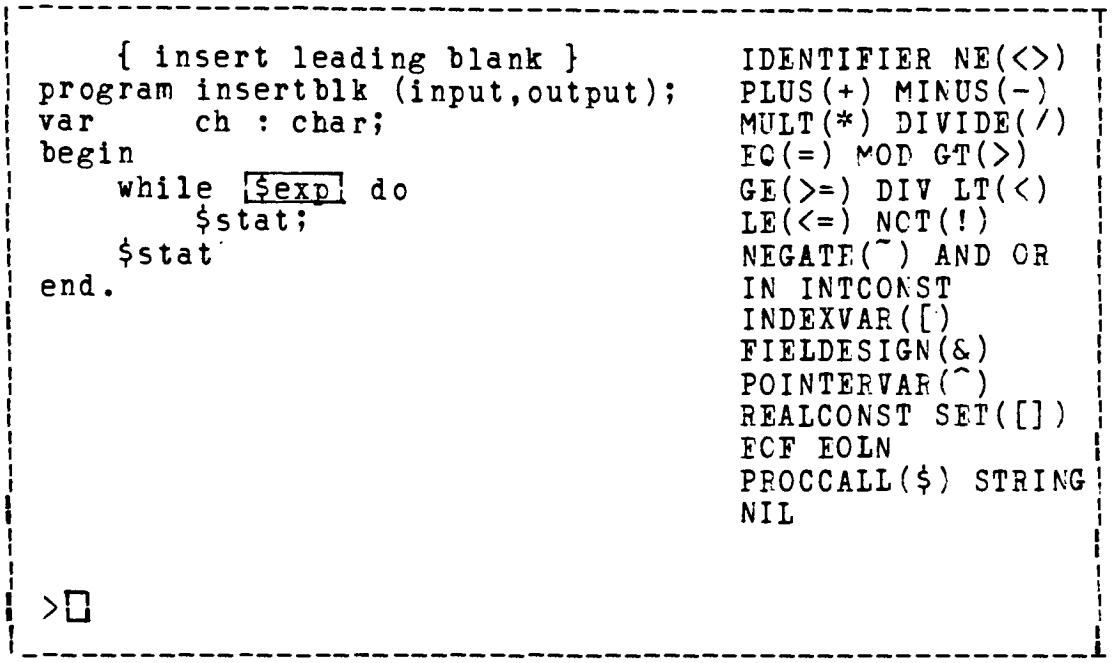
```
    { insert leading blank }                    VARDECL COMVPL
program insertblk (input,output);
var     ch : char;
        $vardecl $procfuncs $statpart.


>
```

If there is an operator called 'IDENT' in the previous class, typing "c" instead of ",c" would cause the command interpreter to accept 'c' as an identifier. This situation can always occur in classes which have the operator IDENT. The prefix ',' or a syncnym must be used to avoid the ambiguity.

In order to begin the statement part of the program, hit '<CR>' twice to end the '$vardecl', skip the "procfuncs", and get to the "$statpart", and type "s" to start the body of the program:



ALOE then generates a "begin" and an "end". The cursor is positioned at '$stat' waiting for a new template to be inserted. A template for the "while" statement is generated by typing the command "wh" followed by "<CR>":



Typing "!" followed by "<CR>" gives the "NOT" statement:

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not [$exp] do
        $stat;
    $stat
end.

>□
```

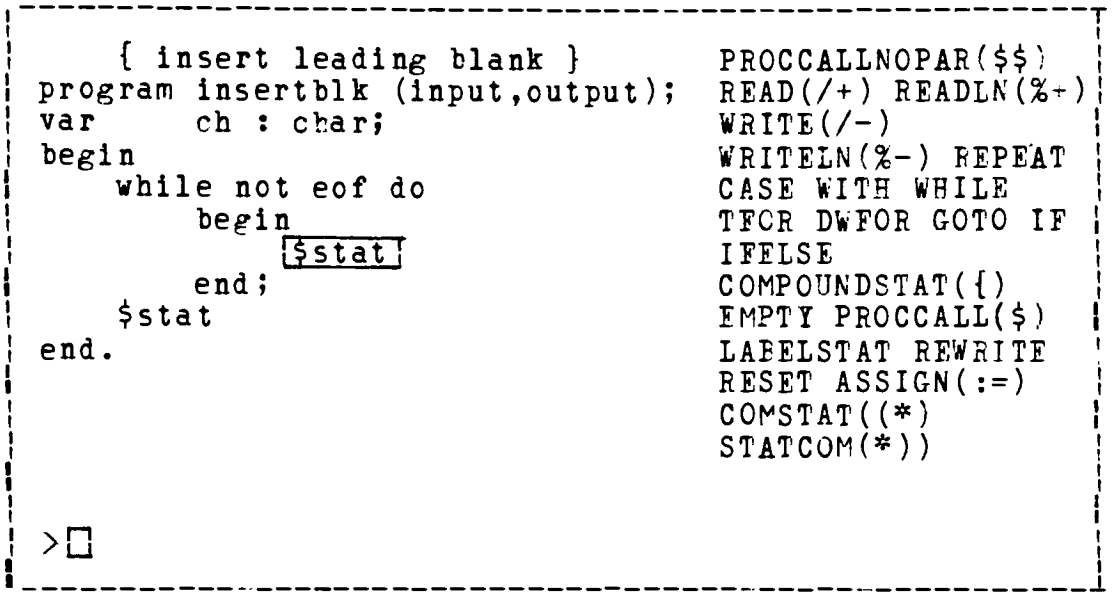
```
IDENTIFIER NE(<>)
PLUS(+) MINUS(-)
MULT(*) DIVIDE(/)
EQ(=) MOD GT(>)
GE(>=) DIV LT(<)
LE(<=) NOT(!)
NEGATE(~) AND OR
IN INTCONST
INDEXVAR([)
FIELDESIGN(&)
POINTERVAR(^)
REALCONST SET([])
EOF EOLN
PROCCALL($) STRING
NIL
```

Typing 'eof' followed by '<CR>' ends the conditional expression:

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof[$fileident] do
        $stat;
    $stat
end.

>□
```

```
EMPTY FILEIDENT
```

Now type "<CR>" to skip the "$fileident" and get to the "$stat". Then input the "{" for the compound statement that forms the body of the "while" statement:



The "begin" and "end" appear on the screen. Typing '/-' for 'write' statement causes:

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write( $out );
            $stat
        end;
    $stat
end.

>
```

```
IDENTIFIER INDEXVAR([)
FIELDESIGN(&)
POINTERVAR(^)
STRING WIDTHFIELD
NE(<>) EQ(=) GT(>)
GE(>=) LT(<)
LE(<=) PLUS(+)
MINUS(-) MULT(*)
DIVIDE(/) MOD DIV
NOT(!) NEGATE(~)
AND OR IN INTCONST
REALCONST EOF EOLN
PROCCALL($)
```

Typing "`,s<CR>`" for a string template followed by "`<SP><CR>`":

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ', $out );
            $stat
        end;
    $stat
end.

>,s
 value: <CR>
```

```
IDENTIFIER INDEXVAR([)
FIELDESIGN(&)
POINTERVAR(^)
STRING WIDTHFIELD
NE(<>) EQ(=) GT(>)
GE(>=) LT(<)
LE(<=) PLUS(+)
MINUS(-) MULT(*)
DIVIDE(/) MOD DIV
NOT(!) NEGATE(~)
AND OR IN INTCONST
REALCONST EOF EOLN
PROCCALL($)
```

If the user only hits '<CR>' when the STRING template is called, ALOE will give an error message for an illegal null string and abort the creation of the string. Typing "<CR>" followed by a "wh" for "while" statement:

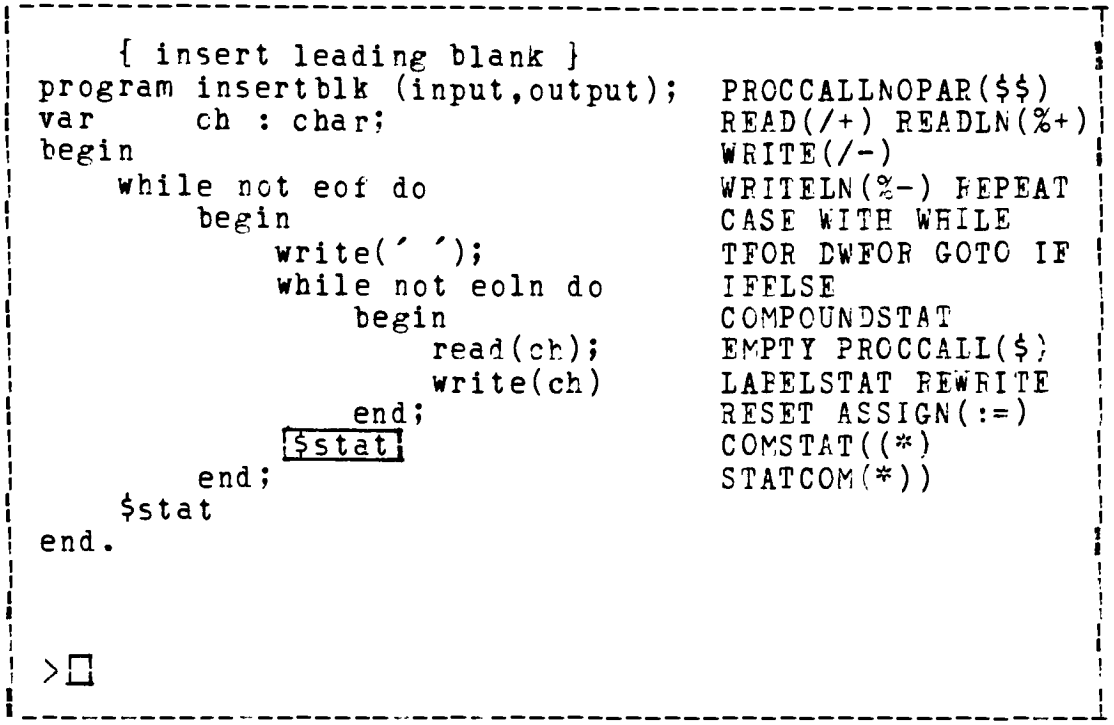
```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ');
            while $exp do
                $stat;
            $stat
        end;
    $stat
end.

>
```

```
IDENTIFIER NE(<>)
PLUS(+) MINUS(-)
MULT(*) DIVIDE(/)
EQ(=) MOD GT(>)
GE(>=) DIV LT(<)
LE(<=) NOT(!)
NEGATE(~) AND OR
IN INTCONST
INDEXVAR([)
FIELDESIGN(&)
POINTERVAR(^)
REALCONST SET([])
EOF EOLN
PROCCALL($) STRING
NIL
```

After typing '!' followed by 'eol', '<CR>', '<CR>', '{' (COMPOUNDSTAT), "/+" (READ), the screen looks as follows:

```
    { insert leading blank }
program insertblk (input,output);        IDENTIFIER
var     ch : char;                       INDEXVAR ([)
begin                                    FIELDESIGN(&)
    while not eof do                     POINTERVAR(^)
        begin
            write(' ');
            while not eoln do
                begin
                    read( $varcomponent );
                    $stat
                end;
            $stat
        end;
    $stat
end.


>
```

Typing "id" (IDENTIFIER) and input "ch" for a variable name followed by "<CR>", "<CR>", "/-" (WRITE), "id", "<CR>", "ch", "<CR>", "<CP>", the screen shows:



If a variable is undeclared, ALOE will give an appropriate errcr message.

Typing "%-" (WRITELN), "<CR>", "<CR>", "%+" (READLN), '<CP>', '<CR>', and '<CR>' will complete the entire program:

```
    { insert leading blank }
program insertblk (input,output);
var     ch: char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.

>□
```

In order to write this tree file into a text file and quit, type "^wu<file name>" and ".q".

3.4. Moving the Cursor

All modifications made to the program occur relative to the current cursor position. Using the cursor keys of the terminal, it is possible to position the cursor wherever insertions and deletions are permitted. The cursor can only be positioned where modifications are allowed.

After every constructive command, ALOE moves the cursor to the next meta node, thus guiding the user in the construction process by placing the cursor in the next available node for expansion. If the next unexpanded node is in an unrelated section of the program, this could cause confusion to the user because ALOE moves the cursor to an undesired location. To solve this problem, use the ".back (^b)" command or '<lf>' instead of '<CR>' after each input command to indicate that ALOE should leave the cursor at the resulting node after the command is applied.

The cursor's motion follows the structure of the tree in a preorder traversal of the the underlying abstract syntax tree. "Cursor-up" ("._OUT") and "cursor-down" ("._IN") move the cursor one level up and down in the tree (to the parent and offspring of the current node). "Cursor-left" and "Cursor-right" move the cursor to the previous and next sibling if they exist; if not, they recursively move it to the next or previous sibling of the parent node. The cursor

'next" and cursor "previous" commands are not symmetric: cursor "next" followed by a cursor "previous" does not necessarily move the cursor back to the original position. 'Back (^b)' moves the cursor back to its previous position but ".terminal" and ".rterminal" move the cursor to the next and previous terminal. Here are other important cursor movement commands.

.find

This command lets the user move rapidly to nodes that are not very close to the current node. On terminal nodes (constants, variables, metas, etc.), a substring match is done with the associated value of the node. On non-terminal nodes the match is done with the names of the corresponding operators and their synonyms. So, it is possible to search for variable names, constant values and names of meta nodes as well as for 'if' or "while" statement.

.window

This command is used to move from one program window to another or to move to and from clipped area windows. Every window has a program cursor associated with it.

The figures below illustrate the use of cursor movement commands.

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.
```

After moving cursor-up ("._OUT"), the cursor is at the

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

After moving cursor-down ('._IN'), the cursor is moved to the first offspring (comment) of the current node:

```
{   { insert leading blank } }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.
```

Since a "comment" is a terminal, cursor-down moves the cursor to the next sibling (prog) of the current node (comment):

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.
```

After moving the cursor-down ('._IN'), the cursor is moved to the first offspring (pgmhead) of a current node (PROG):

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.
```

After moving the cursor-down ("._IN"), the cursor is moved to the first offspring of a "PGMHEAD" which is an identifier:

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.
```

After moving the cursor-right ('._NEXT'), the cursor is moved to the next sibling (idents) of the current node:

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.
```

After moving the cursor right ("._NEXT"), the cursor is moved to the next sibling (block) of the parent of the current node (pgmhead) because there are no more siblings of the current node:

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.
```

After moving the cursor-down, the cursor becomes invisible because there is no label declaration. After moving the cursor 'next' three times, the cursor is at the variable declaration.

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
 begin
     while not eof do
         begin
             write(' ');
             while not eoln do
                 begin
                     read(ch);
                     write(ch)
                 end;
             writeln;
             readln
         end
 end.
```

Because there are no functions or procedures, two cursor movements of "next" will end the "$vardecl" and skip the 'procfuncs':

```
     { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.
```

After moving the cursor-down ("._IN"), the cursor is positioned at the 'statement' which is the offspring of the "STATPART":

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.
```

After moving the cursor-down ("..IN") to the first offspring of the current node:

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.
```

After moving the cursor-right ("._NEXT") to the next sibling (compoundstat):

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.
```

After moving the cursor-down ("._IN"):

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.
```

After moving the cursor-down ('._IN') and cursor-right ('._NEXT'):

```
    { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.
```

After moving the cursor-left ("._PREVIOUS") to the previous sibling (write) of the current node (while):

```
   { insert leading blank }
program insertblk (input,output);
var     ch : char;
begin
    while not eof do
        begin
            write(' ');
            while not eoln do
                begin
                    read(ch);
                    write(ch)
                end;
            writeln;
            readln
        end
end.
```

3.5. Modifying the program

Modifying a program by relocating or changing structural units can be accomplished by using the ".clip", ".delete", ".insert", ".replace", ".nest", and ".transform" commands.

.clip and .insert

These are used to copy and move subtrees. Clipped subtrees are kept on a separate clipped area where they can be inspected and edited.

.delete and .replace

These are used to delete subtrees. The ".replace" command leaves a meta node in the place of the deleted subtree.

.nest and .transform

These are very important because they make editing structures much easier and contibute towards making structured editing much more attractive. The ".nest" command makes a new subtree in the place of the current one with the current subtree as offspring of the new root (provided that the resulting subtree is a legal one). Figure 3.1 shows the effect of nesting an assignment statement into an "while" statement. Figure 3.2 shows the nesting of an addition into

a multiplication.

The ".transform" command changes the operator of the current subtree root provided that the transformation is a legal one. Figure 3.3 shows the effect of transforming a 'while' statement into an "if" statement. Figure 3.4 shows the transformation of an "if" statement into an "if then else" statement. type "e".

```
if $exp then
     value := value + 1;

Typing ^x^n <while>:

if $exp then
     while $exp do
          value := value + 1;
```

Figure 3.1. Nesting an "assignment" statement into a 'while' statement

```
if $exp then
     a:= b + c ;

Typing ^x^n <*>:

if $exp then
     a = (b + c) * $exp
```

Figure 3.2. Nesting an addition into a multiplication

```
while a = b do
     a:= (b + c) * d ;

Typing ^t <if>:

if a = b then
     a:= (b + c) * d ;
```

Figure 3.3. Transformation of a "while" statement into an "if" statement

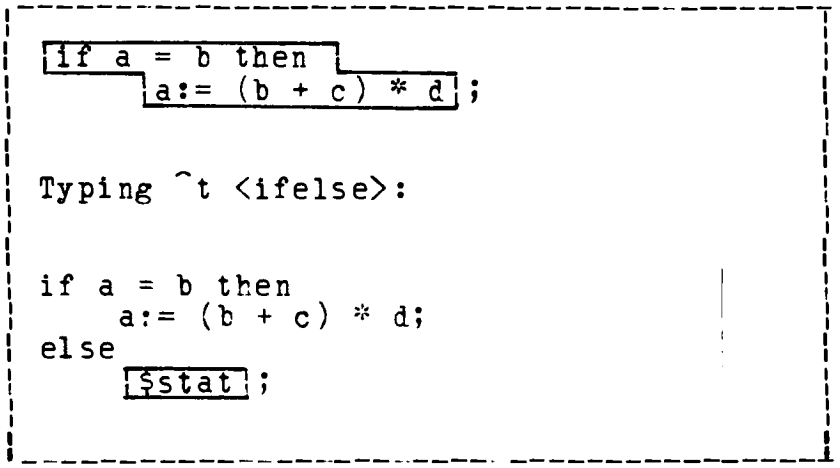


Figure 3.4. Transformation of an "if" statement into an 'if then else' statement

CHAPTER 4

SUMMARY

A syntax-directed editor for Pascal is a structure editor where the emphasis is placed in the constructions of the language. The aspect of the design is the separation of abstract syntax and concrete representation.

Chapter 1 provided the discussion and comparison of the text-oriented and syntax-directed editor. The grammar description and generation of syntax-directed editor for Pascal was covered in Chapter 2. Chapter 3 described the user interface: how to create, move the cursor, and modify a program.

4.1. Missing Features

Some important concepts are missing from the design of the grammatical description. It is lacking lists which contain at least one element (non-empty list). Non-empty lists are necessary to be able to avoid syntactic inaccuracies in certain language constructs. For the Pascal syntax-directed editor, an empty list is replaced by an error message so that the user can back up and correct it.

Some of the goals of the user interface are to make it possible for the user to interact with the editor in terms

of the structure of the program being edited, and minimization of the work that needs to be done by the user. It turns cut to be more effort when the 'comment' operators are added to every class that should have a comment statement. For example, comments at the top cf the program, before the constant declaration, type declaration, variable declaration, etc. Comments have no internal structures and they are designed to be thrown away by the lexical analyzer. Detection of undefined variables and type checking also increase the user's work since the user has to invoke the command "IDENTIFIER" or "TYPEIDENT" before typing the identifier name or type identifier, rather than just typing in the identifier name. Since comments are important to good programming style and practice, they are still in the grammar.

The text editor command ('x't) which are used to edit the character constant or string by using a text editor 'ex' does not work properly. It causes the last character of the line being edited to be moved to the next line, which might cause an error.

The pretty-printing of record types works only when there are five or fewer levels of records because the ALCE system is designed to have only four levels of stack for push and pop markers that are used to remember column positions for formatting in the unparsing scheme.

4.2. Enhancements

Here are some ideas that could be considered for future research. Some of the synonyms for editing commands could be changed to things that are more natural for the user. Synonyms for the constructive commands can be changed easily by merely replacing them in the grammar for the language. Synonyms for different operators could be the same if the operators are in different classes, but a problem may occur when the ".find" command is used to search for an operator if the user enters a synonym as a searching string. The display package could also be modified to interface to terminals other than VT100's and C100's. To eliminate the inappropriate line update or cursor highlight, the user has to redisplay the screen by using "^l". It would be better to do more type checking so as to minimize the effort of the user; an extension of the editor for Pascal to understand language semantics (i.e., to have it perform context sensitive processing) might be one approach. Extended commands to communicate with other tools of the environment (like ".compile" or ".run" commands) would improve the quality and usefulness of a syntax-directed editor by eliminating the step of unparsing a tree file into a text file. A parsing interface could also be developed to allow the editing of programs created with an ordinary text editor to be changed using the syntax-directed editor.

BIBLIOGRAPHY

[Bars81] Barstow, D., "Overview of a Display-Oriented Editor for INTERLISP", Proc. of IJCAI-81, August 1981: pp. 927-929.

[Conw76] Conway, R. "Primer on Disciplined Programming Using PL/CS", Tech. Rept, Department of Computer Science, Cornell Univ. NY, 1976: 76-293.

[Donz75] Donzeau-Gouge, V.et.al., "Structure Oriented Program Editor: a First Step Towards Computer Assisted Programming", Proc. Inter. Computing Symp., Antibes, 1975.

[Donz80] Donzeau-Gouge, V, Huet, Gerard, Kahn, Gilles and Lang, Bernard, "Programming Environments Based on Structured Editors: The MENTOR Experience", Presented at the Workshop on Programming Environments in Ridgefield, CT. June 1980.

[Fras81] Fraser, W. Christopher, "Syntax-Directed Editing of General Data Structures", ACM SIGPLAN Notices, 16(6) June 1981: pp. 17-21.

[Jens74] Jensen, K. and Wirth, Niklaus, Pascal User Mannual and Report, Springer-Verlag, New York Heidelberg Berlin, 1974.

[Kern78] Kernighan, B.W. and Ritchie, D.M., Prentice-Hall Software Series: The C Programming Language, Prentice-Hall, 1978.

[Medi81a] Medina-Mora, R. and Fieler, P., "An Incremental Pregramming Environment", IEEE Trans on Soft. Eng., 7(5) September 1981: pp 472-482.

[Medi81b] Medina-Mora, R. and Notkin, S.D., ALOE Users' and Implementators Guide, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., November 1981.

[Morr81] Morris, M. Joseph and Schwartz D. Mayer, "The Design of a Language-Directed Editor for Block Structured Languages", ACM SIGPLAN Notices, 16(6) June 1981: pp. 35-71.

[Shap81] Shapiro, E. et. al., "PASES: a Programming Environment for PASCAL", ACM SIGPLAN Notices, 16(8) August 1981: pp. 50-57.

[Stro81] Stromfors, O. and Jones, jo L., "The Implementation and Experiences of a Structure-Oriented Text Editor", ACM SIGPLAN Notices, 16(6) June 1981: pp. 22-27.

[Teit78] Teitelman, Warren, 'INTERLISP Reference Manual', Xerox PARC Tech. Rep., September 1978.

[Teit76] Teitelbaum, T., 'A formal syntax for PL/CS', Tech Rept, Department of Computer Science, Cornell Univ., Ithaca, NY, June 1976: pp 76-281.

[Teit81a] Teitelbaum, T. and Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", Comm. ACM, 24(9) September 1981: pp. 563-573.

[Teit81b] Teitelbaum, T., Reps, T., and Horwitz, S., "The Why and Wherefore of the Cornell Program Synthesizer", ACM SIGPLAN Notices, 16(6) June 1981: pp 8-16.

[Wate82] Waters, C.R., 'Program Editors Should Not Abandon Text Oriented Commands", ACM SIGPLAN Notices, 17(7) July 1982: pp 39-46.

[Wilc76] Wilcox, T.R., Davis, A.M., and Tindall, M.H., "The Design and Implementation of a Table Driver, Interactive Diagnostic Programming System", Comm. ACM, 19(11) November 1976: pp. 609-616.

APPENDIX A

ALOE GRAMMAR FOR PASCAL

```
/* terminals */

NIL  =
     {s} | action: <none> | synonym: <none> | lex: <none> |
     (Ø) "nil";

REAL  =
     {s} | action: <none> | synonym: <none> | lex: <none> |
     (Ø) "real";

INTEGER  =
     {s} | action: <none> | synonym: <none> | lex: <none> |
     (Ø) "integer";

BOOLEAN  =
     {s} | action: <none> | synonym: <none> | lex: <none> |
     (Ø) "boolean";

CHAR  =
     {s} | action: <none> | synonym: <none> | lex: <none> |
     (Ø) "char";

INTCONST  =
     {i} | action: <none> | synonym: <none> | lex: lexinteger
     (Ø) "@c";

REALCONST  =
     {r} | action: <none> | synonym: <none> | lex: lexreal |
     (Ø) "@c";

IDENT  =
     {v} | action: aIDENT | syncnym: <none> | lex: lexvariable
     (Ø) "@s";

EMPTY  =
     {s} | action: <none> | synonym: <none> | lex: <none> |
     (Ø) "";

COMMENT  =
     {e} | action: <none> | synonym: <none> | lex: <none> |
     (Ø) "@>{ @c }"
```

```
     (1) "@>(* @c *)";

GOTO  =
     {i} ¦ action: <none> ¦ synonym: <none> ¦ lex: lexinteger
     (Ø) "goto @c";

STRING  =
     {e} ¦ action: aSTR ¦ synonym: <none> ¦ lex: <none> ¦
     (Ø) "'@c'";

TEXT  =
     {s} ¦ action: <none> ¦ synonym: <none> ¦ lex: <none> ¦
     (Ø) "text";




/* nonterminals */

PROGRAM  =
     prog ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Filencde ¦
     (Ø) "@1";

PROG  =
     pgmhead block ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@n@1@2.";

PGMHEAD  =
     ident idents ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "program @1 (@2);";

IDENTS  =
     <ident> ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@Ø,@e-** Inserted identifier **-";

BLOCK  =
     labeldecl constdecl typedecl vardecls procfuncs
     statpart ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@1 @2 @3 @4 @5 @6";

PROCFUNCS  =
     <procfunc> ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
```

```
     (Ø) "@Ø";

LABELPART  =
     llabel ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@nlabel   @1;";

LLABEL  =
     <intconst> ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) '@Ø,@e-** Inserted number **-';

CONSTDECL  =
     lconstdef ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@nconst   @1";

CONSTDEF  =
     ident constant ¦ action: <none> ¦ synonym: ';' ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@1 = @2;";

STATPART  =
     statement ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) '@nbegin@+@n@1@-@nend';

LSTATS  =
     <stat> ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@Ø;@n";

IF  =
     exp stat ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "if @1 then @+@n@2@-";

PLUS  =
     exp exp ¦ action: <none> ¦ synonym: "+" ¦
     precedence: 3 ¦ Non-filenode ¦
     (Ø) '@1 + @2';

MINUS  =
     exp exp ¦ action: <none> ¦ synonym: "-" ¦
     precedence: 3 ¦ Non-filenode ¦
     (Ø) "@1 - @2";

MULT  =
     exp exp ¦ action: <none> ¦ synonym: '*' ¦
     precedence: 4 ¦ Non-filenode ¦
     (Ø) "@1 * @2";
```

```
DIVIDE  =
     exp exp ¦ action: <none> ¦ synonym: '/' ¦
     precedence: 4 ¦ Non-filenode ¦
     (Ø) "@1 / @2";

DIV  =
     exp exp ¦ action: <none> ¦ synonym: <none> ¦
     precedence: 4 ¦ Non-filenode ¦
     (Ø) '@1 div @2';

EQ  =
     exp exp ¦ action: <none> ¦ synonym: "=" ¦
     precedence: 2 ¦ Non-filenode ¦
     (Ø) "@1 = @2";

LCONSTDEF  =
     <constdef> ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@Ø@n@>@>@e-** Expected identifier **-";

TYPEDEFPART  =
     typedefs ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) '@ntype    @1';

TYPEDEFS  =
     <typedef> ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@Ø@n@>@>@e-** Expected identifier **-";

TYPEDEF  =
     ident type ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@1 = @2;";

SIMPLETYPE  =
     simpletype ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) '@1';

SUBRANGE  =
     constant constant ¦ action: <none> ¦ synonym: "}" ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@1..@2";

SCALAR  =
     idents ¦ action: <none> ¦ synonym: '{' ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "(@1)";

POINTERTYPE  =
```

```
      identifier ¦ action: <none> ¦ synonym: '^' ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '^@1';

MOD   =
      exp exp ¦ action: <none> ¦ synonym: <none> ¦
      precedence: 4 ¦ Non-filenode ¦
      (Ø) "@1 mod @2";

GT    =
      exp exp ¦ action: <none> ¦ synonym: '>' ¦
      precedence: 2 ¦ Non-filenode ¦
      (Ø) "@1 > @2";

GE    =
      exp exp ¦ action: <none> ¦ synonym: ">=" ¦
      precedence: 2 ¦ Non-filenode ¦
      (Ø) '@1 >= @2';

PACKSTRUCT  =
      unpackstruct ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "packed @1";

ARRAYTYPE  =
      lindextype type ¦ action: <none> ¦ synonym: '[' ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "array [@1] of @2";

LINDEXTYPE  =
      <indextype> ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '@Ø,@e-** Inserted identifier **-';

RECORDTYPE  =
      fieldlist ¦ action: <none> ¦ synonym: "()" ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@p1record@n@g1  @p2@1@n@g1@r1end@r2";

FIXPART  =
      lrecordsection ¦ action: <none> ¦ synonym: '<*' ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@1";

LRECORDSECTION  =
      <recordsection> ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '@Ø;@n';

RECORDSECTION  =
      lfieldident type ¦ action: <none> ¦ synonym: <none>
```

```
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@g2@1 : @2";

LFIELDIDENT =
     <fieldident> ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@Ø,";

FIELDIDENT =
     ident ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) '@1 : ';

VARIANTPART =
     tagfield type lvariant ¦ action: <none> ¦ synonym: ">*" ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@g2case @pØ@1 @2 of@n@3@rØ";

LVARIANT =
     <variant> ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@Ø;@n";

VARIANT =
     caselabelist fieldlist ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) '@gØ@1 : (@p2@2)@g2@r2';

FIXVARIANT =
     fixpart variantpart ¦ action: <none> ¦ synonym: "<>" ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@1;@n@2";

SETTYPE =
     basetype ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "set of @1";

FILETYPE =
     type ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) 'file of @1';

CASELABELIST =
     <caselabel> ¦ action: <none> ¦ synonym: <none> ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "@Ø,";

LT =
     exp exp ¦ action: <none> ¦ synonym: '<' ¦
     precedence: 2 ¦ Non-filenode ¦
```

```
     (Ø) "@1 < @2";

LE   =
     exp exp ¦ action: <none> ¦ synonym: '<=' ¦
     precedence: 2 ¦ Non-filenode ¦
     (Ø) "@1 <= @2";

NE   =
     exp exp ¦ action: <none> ¦ synonym: "<>" ¦
     precedence: 2 ¦ Non-filenode ¦
     (Ø) '@1 <> @2';

NOT  =
     exp ¦ action: <none> ¦ synonym: "!" ¦
     precedence: 5 ¦ Non-filenode ¦
     (Ø) "not @1";

OR   =
     exp exp ¦ action: <none> ¦ synonym: <none> ¦
     precedence: 3 ¦ Non-filenode ¦
     (Ø) "@1 or @2";

AND  =
     exp exp ¦ action: <none> ¦ synonym: <none> ¦
     precedence: 4 ¦ Non-filenode ¦
     (Ø) '@1 and @2';

SET  =
     setelement ¦ action: <none> ¦ synonym: "[]" ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) "[@1]";

IN   =
     ivarcomponent set ¦ action: <none> ¦ synonym: <none> ¦
     precedence: 2 ¦ Non-filenode ¦
     (Ø) "@1 in @2";

INDEXVAR  =
     varcomponent exps ¦ action: <none> ¦ synonym: "[" ¦
     precedence: <none> ¦ Non-filenode ¦
     (Ø) '@1[@2]';

EXPS  =
     <exp> ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@Ø,@e-** Inserted identifier **-";

FIELDESIGN  =
      varcomponent fieldid ¦ action: <none> ¦ synonym: '&'
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@1.@2";
```

```
POINTERVAR  =
      varcomponent ¦ action: <none> ¦ syncnym: '^' ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@1^";

NORANGESET  =
      <element> ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '@Ø, ';

RANGESET  =
      element element ¦ action: <none> ¦ synonym: <none>
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@1..@2";

ASSIGN  =
      varcomponent exp ¦ action: <none> ¦ synonym: ':=' ¦
      precedence: 1 ¦ Non-filenode ¦
      (Ø) "@1 := @2";

LACTPARAM  =
      <actparam> ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '@Ø, ';

EXP  =
      exp ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@1";

VARDECLS  =
      lvardecl ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@nvar     @1";

LVARDECL  =
      <vardecl> ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '@Ø@n@>@>@e-** Expected identifier **-';

VARDECL  =
      idents type ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@1 : @2;";

WHILE  =
      exp stat ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "while @1 do@+@n@2@-";

TFOR  =
```

```
      identifier exp exp stat ¦ action: <none> ¦
      synonym: <none> ¦ precedence: <none> ¦ Non-filenode ¦
      (Ø) 'for @1 := @2 to @3 do@+@n@4@-';

DWFOR =
      identifier exp exp stat ¦ action: <none> ¦
      synonym: <none> ¦ precedence: <none> ¦ Non-filenode ¦
      (Ø) "for @1 := @2 downto @3 do@+@n@4@-";

REPEAT =
      lrepeatstat untilexp ¦ action: <none> ¦ synonym: <none>
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "repeat@+@n@1@-@nuntil@n@+@>@2@-";

LREPEATSTAT =
      <stat> ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '@Ø;@n@e;';

WITH =
      lrecordvar stat ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "with @1 do@+@n@2@-";

LRECORDVAR =
      <recordvar> ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@Ø,";

CASE =
      exp lcaselement ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) 'case @1 of@+@n@2@-@nend';

LCASELEMENT =
      <caselement> ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@Ø;@n";

CASELEMENT =
      caselabelist stat ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@1 : @2";

READ =
      lvarcomponent ¦ action: <none> ¦ synonym: "/+" ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) 'read(@1)';

LVARCOMPONENT =
      <varcomponent> ¦ action: <none> ¦ synonym: <none> ¦
```

```
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@Ø,@e-** Inserted identifier **-";

WRITELN =
      outputlist ¦ action: <none> ¦ synonym: '%-' ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "writeln@1";

OUTPUTLIST =
      listout ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '(@1)';

LISTOUT =
      <out> ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@Ø,@e-** Inserted identifier **-";

WIDTHFIELD =
      wvar ovalue ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@1@2";

OVALUE =
      firstval secondval ¦ action: <none> ¦
      synonym: <none> ¦ precedence: <none> ¦ Non-filenode
      (Ø) '@1@2';

FVAL =
      fval ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) ":@1";

SVAL =
      sval ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) ":@1";

WRITE =
      listout ¦ action: <none> ¦ synonym: "/-" ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) 'write(@1)';

READLN =
      inputlist ¦ action: <none> ¦ synonym: "%+" ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "readln@1";

INPUTLIST =
      lvarcomponent ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
```

```
      (Ø) "(@1)";

COMPOUNDSTAT  =
      stats ¦ action: <none> ¦ synonym: "{" ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) 'begin@+@n@1@-@nend';

IFELSE  =
      exp state stat ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "if @1 then@+@n@2@-@nelse@+@n@3@-";

PROCED  =
      prccheading block ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filencde ¦
      (Ø) "@n@1;@2;";

PROCHEAD  =
      ident ¦ action: <none> ¦ synonym: ")" ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) 'procedure @1';

PROCFORMALHEAD  =
      ident lformalparam ¦ action: <none> ¦ synonym: "(" ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "procedure @1 (@2)";

LFORMALPARAM  =
      <formalparam> ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@Ø; ";

PARAMGROUP  =
      idents type ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '@1 : @2';

VARPARAMGROUP  =
      paramgroup ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "var @1";

FUNCT  =
      funcheading block ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@n@1;@2;";

FUNCNOPARAM  =
      ident type ¦ action: <none> ¦ synonym: ">>" ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) 'function @1 : @2';
```

```
FUNCPARAM  =
      ident lformalparam type ¦ action: <none> ¦
      synonym: '<<' ¦ precedence: <none> ¦ Non-filenode ¦
      (Ø) "function @1 (@2) : @3";

PROCCALL  =
      procname lactparam ¦ action: <none> ¦ synonym: '$' ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@1(@2)";

PROCCALLNOPAR  =
      procname ¦ action: <none> ¦ synonym: "$$" ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '@1';

REWRITE  =
      identifier ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "rewrite(@1)";

RESET  =
      identifier ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "reset(@1)";

EOF  =
      fileident ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) 'eof@1';

FILEIDENT  =
      identifier ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "(@1)";

NEGATE  =
      exp ¦ action: <none> ¦ synonym: '~' ¦
      precedence: 5 ¦ Non-filenode ¦
      (Ø) "-@1";

COMPROCFUNC  =
      comments procfunc ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '@n@1 @2';

COMMENTS  =
      <comment> ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@Ø@n";

COMCONST  =
```

```
      comments constdecl ¦ action: <none> ¦ synonym: <none>
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@n@1 @2";

COMSTATPART  =
      comments statpart ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '@n@1 @2';

COMTYPE  =
      comments typedecl ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@n@1 @2";

COMVAR  =
      comments vardecls ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@n@1 @2";

CCOM  =
      constdef comment ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '@1@2';

COMTY  =
      typedef comment ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@1@2";

COMVBL  =
      vardecl comment ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@1@2";

STATCOM  =
      stat comment ¦ action: <none> ¦ synonym: "*)" ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '@1@2';

EOLN  =
      fileident ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "eoln@1";

LABELSTAT  =
      intconst lbstat ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@1 : @2";

COMPROG  =
      comments prog ¦ action: <none> ¦ synonym: <none> ¦
```

```
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '@1@2';

COMSTAT  =
      comment stat ¦ action: <none> ¦ synonym: "(*" ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@1@n@2";

FORWARD  =
      forward ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@n@1; forward;";

PROCPARAMS  =
      idents ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) 'procedure @1';

TYPEIDENT  =
      ident ¦ action: aTYPE ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@1";

IDENTIFIER  =
      ident ¦ action: aVAR ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) "@1";

FIELDID  =
      identifier ¦ action: <none> ¦ synonym: <none> ¦
      precedence: <none> ¦ Non-filenode ¦
      (Ø) '@1';




/* classes */


prog  =
     PROG COMPROG ;

pgmhead  =
     PGMHEAD ;

block  =
     BLOCK ;
```

```
ident  =
     IDENT ;

labeldecl  =
     EMPTY LABELPART ;

constdecl  =
     EMPTY CONSTDECL COMCONST ;

statpart  =
     STATPART COMSTATPART ;

statement  =
     LSTATS ;

llabel  =
     LLABEL ;

intconst  =
     INTCONST ;

lconstdef  =
     LCONSTDEF ;

constdef  =
     CONSTDEF CCOM ;

stats  =
     LSTATS ;

stat  =
     PROCCALLNOPAR READ READLN WRITE WRITELN REPEAT CASE
     WITH WHILE TFOR DWFOR GOTO IF IFELSE COMPOUNDSTAT EMPTY
     PROCCALL IABELSTAT REWRITE RESET ASSIGN COMSTAT STATCOM ;

exp  =
     IDENTIFIER NE PLUS MINUS MULT DIVIDE EQ MOD GT GE DIV
     LT LE NOT NEGATE AND OR IN INTCONST INDEXVAR FIELDESIGN
     POINTERVAR REALCONST SET EOF EOLN PROCCALL STRING NIL ;

idents  =
     IDENTS ;

constant  =
     IDENTIFIER STRING INTCONST REALCONST NIL NEGATE ;

typedecl  =
     EMPTY TYPEDEFPART COMTYPE ;

typedefs  =
     TYPEDEFS ;
```

```
typedef  =
     TYPEDEF COMTY ;

type  =
     TYPEIDENT SCALAR SUBRANGE PACKSTRUCT ARRAYTYPE
     RECORDTYPE SETTYPE FILETYPE POINTERTYPE BOOLEAN REAL
     INTEGER CHAR TEXT ;

simpletype  =
     IDENT SCALAR SUBRANGE CHAR ;

unpackstruct  =
     ARRAYTYPE RECORDTYPE SETTYPE FILETYPE ;

lindextype  =
     LINDEXTYPE ;

indextype  =
     TYPEIDENT SCALAR SUBRANGE CHAR ;

fieldlist  =
     FIXPART FIXVARIANT VARIANTPART EMPTY ;

lrecordsection  =
     LRECORDSECTION ;

recordsection  =
     RECORDSECTION ;

lfieldident  =
     LFIELDIDENT ;

fieldident  =
     IDENT ;

tagfield  =
     FIELDIDENT EMPTY ;

lvariant  =
     LVARIANT ;

variant  =
     EMPTY VARIANT ;

caselabelist  =
     CASELABELIST ;

fixpart  =
     FIXPART ;

variantpart  =
```

```
     VARIANTPART ;

basetype  =
     SIMPLETYPE ;

caselabel  =
     IDENTIFIER INDEXVAR FIELDESIGN POINTERVAR STRING
     INTCONST REALCONST ;

varcomponent  =
     IDENTIFIER INDEXVAR FIELDESIGN POINTERVAR ;

setelement  =
     EMPTY IDENTIFIER RANGESET NORANGESET ;

set  =
     IDENTIFIER SET ;

exps  =
     EXPS ;

actparam  =
     IDENTIFIER EXP INDEXVAR FIELDESIGN POINTERVAR STRING
     INTCONST REALCONST ;

lactparam  =
     LACTPARAM ;

vardecls  =
     EMPTY VARDECLS COMVAR ;

lvardecl  =
     LVARDECL ;

vardecl  =
     VARDECL COMVBL ;

lrecordvar  =
     LRECORDVAR ;

recordvar  =
     IDENTIFIER INDEXVAR FIELDESIGN POINTERVAR ;

lcaselement  =
     LCASELEMENT ;

lvarcomponent  =
     LVARCOMPONENT ;

outputlist  =
     EMPTY OUTPUTLIST ;
```

```
listout  =
     LISTOUT ;

out  =
     IDENTIFIER INDEXVAR FIELDESIGN POINTERVAR STRING
     WIDTHFIELD NE EQ GT GE LT LE PLUS MINUS MULT DIVIDE
     MOD DIV NOT NEGATE AND OR IN INTCONST REALCONST EOF
     EOLN PROCCALL ;

ovalue  =
     OVALUE ;

firstval  =
     EMPTY FVAL ;

secondval  =
     EMPTY SVAL ;

inputlist  =
     EMPTY INPUTLIST ;

procheading  =
     PROCHEAD PROCFORMALHEAD ;

forward  =
     PROCHEAD PROCFORMALHEAD FUNCPARAM FUNCNOPARAM ;

lformalparam  =
     LFORMALPARAM ;

formalparam  =
     PARAMGROUP VARPARAMGROUP FUNCPARAM FUNCNOPARAM
     PROCFORMALHEAD PROCHEAD PROCPARAMS ;

paramgroup  =
     PARAMGROUP ;

funcheading  =
     FUNCNOPARAM FUNCPARAM ;

procname  =
     IDENT ;

caselement  =
     CASELEMENT ;

comment  =
     COMMENT ;

fileident  =
     EMPTY FILEIDENT ;
```

```
element  =
     IDENTIFIER INTCONST STRING ;

procfuncs  =
     EMPTY PROCFUNCS ;

comments  =
     COMMENTS ;

procfunc  =
     PROCED FUNCT FORWARD COMPROCFUNC ;

fval  =
     IDENTIFIER INTCONST ;

sval  =
     IDENTIFIER INTCONST ;

ivarcomponent  =
     IDENTIFIER INDEXVAR FIELDESIGN POINTERVAR PROCCALL
     STRING EXP ;

lbstat  =
     PROCCALLNOPAR READ READLN WRITE WRITELN REPEAT CASE
     WITH WHILE TFOR DWFOR GOTO IF IFELSE COMPOUNDSTAT
     EMPTY PROCCAL REWRITE RESET ASSIGN COMSTAT STATCOM ;

wvar  =
     IDENTIFIER INDEXVAR FIELDESIGN POINTERVAR STRING NE
     EQ GT GE IT LE PLUS MINUS MULT DIVIDE MOD DIV NOT
     NEGATE AND OR IN INTCONST REALCONST EOF EOLN PROCCALL ;

identifier  =
     IDENTIFIER ;

fieldid  =
     IDENTIFIER FIELDID ;

lrepeatstat  =
     LREPEATSTAT ;

state  =
     PROCCALLNOPAR READ READLN WRITE WRITELN REPEAT
     CASE WITH WHILE TFOR DWFOR GOTO IF IFELSE COMPOUNDSTAT
     PROCCALL LABELSTAT REWRITE RESET ASSIGN COMSTAT
     STATCOM ;
untilexp  =
     IDENTIFIER NE PLUS MINUS MULT DIVIDE EQ MOD GT GE DIV
     LT LE NOT NEGATE AND OR IN INDEXVAR FIELDESIGN
     POINTERVAR EOP EOLN PROCCAIL ;
```

An example of an Environment-Specific routine:

/* this routine replaces the default lexical routine for variables. The default routine accepts alphanumerics and underscores as legal characters, while this routine permits only alphanumerics. */

```
#include <ctype.h>
char *lexvariable(inittok, buffer, start, finish)
int inittok;
char *buffer, **start, **finish;

{
        char  name[100];
        if (!isalpha (*buffer))
                return buffer;
        *start = buffer;
        buffer++;
        for (; isalnum (*buffer); buffer++);
        *finish = buffer;
        strncpy (name, *start, *finish - *start);
        if (isreserved (name))
                return *start;
        else
                return buffer;
}
```

Action Routines for the Pascal ALOE:

/* The action routine for IDENT checks the identifier name to see if it is a reserved word. If it is, an error message is printed. For the RENAME command, the error message is passed to the user. For the CREATE, the creation is also aborted. */

```
#include "cALOELIB.h"
#include "GANDALF.h"
#include "key.h"
aIDENT(thisnode, actkind)
struct tnode *thisnode;
int actkind;
{
  char *iden;
  switch (actkind) {
     case CREATE:
       iden = ((struct tnodev *)thisnode)->key->myname;
       if (isreserve (iden))
       {
       printerror
       ("***** WARNING ***** error in keyword identifier name");
       return (-2);   /* abort the creation */
       }
     case RENAME:
       iden = ((struct tnodev *)thisnode)->key->myname;
       if (isreserve (iden))
       {
       printerror
       ("***** WARNING *****  error in keyword identifier name")
       return (-1);   /* continue but still report error */

       }
     default:
       break;
     }
     return NIL;
}
```

/* This routine does a binary search for a keyword as defined in a keyword table */

```
#define NKEYS 36
int isreserve(name)
char *name;
{
     int low, high, mid, cond;
        low = 0;
        high = NKEYS - 1;
        while (low <= high)
        {
                mid = (low + high)/2;
                if ((cond = strcmp(name, keytab[mid])) < 0)
                    high = mid - 1;
                else
                    if (cond > 0)
                       low = mid + 1;
                else
                    return (TRUE);
        }
        return (FALSE);
}
```

```
/* keyword table */

static char *keytab[] = {
                    "and",
                    "array",
                    "begin",
                    "case",
                    "const",
                    "div",
                    "do",
                    "downto",
                    "else",
                    "end",
                    "file",
                    "for",
                    "forward",
                    "function",
                    "goto",
                    "if",
                    "in",
                    "label",
                    "mod",
                    "nil",
                    "not",
                    "of",
                    "or",
                    "packed",
                    "procedure",
                    "program",
                    "record",
                    "repeat",
                    "set",
                    "then",
                    "to",
                    "type",
                    "until",
                    "var",
                    "while",
                    "with"
                };
```

/* The action routine for STRING checks to see that the character is empty. If it is, it prints out an error message and aborts the creation. */

```
#include "cALOELIB.h"
#include "GANDALF.h"

aSTR(thisnode, actkind)
struct tnode *thisnode;
int actkind;
{
   char *string;
   switch (actkind) {
      case CREATE:
        string = ((struct tnodec *)thisnode)->ctname;
        if (*string == ' ')
        {
            printerror
            ("***** error in empty character constant *****");
            return (-2);   /* abort the creation */
        }
      default:
            break;
        }
   return NIL;
}
```

/* This is an action routine that checks for an undeclared variable. The error will be reported if the variable is undefined, and ALOE will abort the creation. */

```
#include "p5.infop"
#include "cALOELIB.h"
#include "GANDALF.h"
#define Tnodev(node) ((struct tnodev *)node)
#define firstson(node) getson(node,0)
struct tnode *aVAR (node,act)
struct tnode *node;
int act;
{
  char *string;
  switch (act) {
    case CREATE:
     chkmake(iIDENT,firstson(node),Tnodev(node)->key->myname);
     if (Tnodev(firstson(node))->key->refcount < 2 )
     {
         printerror ( ' WARNING **** undefined variable ');
         return ((struct tnode *)-2);
     }
    default:
     break;
    }
    return ((struct tnode *)NIL);
}
```

/* This is an action routine that checks for an undeclared type. The error will be reported if the type is undefined, and ALOF will abort the creation. */ occurred. */

```
#include "p5.infop"
#include "cALOFLIB.h"
#include "GANDALF.h"
struct tnode *aTYPE (node,act)
struct tnode *node;
int act;
{
  switch (act) {
   case CREATE:
     chkmake
     (iIDENT,getson(node,0),((struct tnodev *)node)->key->myname);
     if (((struct tnodev *)(getson(node,0)))->key->refcount < 2)
     {
        printerror ( " WARNING **** undefined type ");
        return ((struct tnode *)-2);
     }
  default:
     break;
  }
  return ((struct tnode *)NIL):
}
```

APPENDIX B

ALOE EDITING COMMANDS

Editing commands are invoked by typing a dot ('.') followed by the name of the command and a <CR>. Only enough characters to designate the command unambiguously are required. All editing commands also have synonyms that are entered without either the dot or <CR>; the synonyms for each command are shown in parenthesis following the command name. A "^" stands for a "control" character.

B.1. Cursor Movement

Cursor-in

._IN (<cursor-down)

Moves the cursor into the first legal offspring of the current node according to current unparsing scheme. Cursor-in automatically does a "cursor-next" if it is at a terminal or non-visible node.

Cursor-out

._OUT (<cursor-up>)

Moves the cursor to the parent of the current node.

Cursor-next

._NEXT (cursor-right)

Moves the cursor to the next sibling of the current node (if one is defined) according to the current unparsing scheme and the setting of the "cursor-follow" mode. If no sibling is defined, the cursor is then moved to the next sibling of the parent of the current node, recursively. If the current node is the last in the tree (as defined in pre-order) then the command has no effect.

Cursor-previous

._PREVIOUS (<cursor-left>)

Moves the cursor to the previous sibling of the current node (if one is defined) according to the current unparsing scheme and the setting of the "cursor-follows" mode. If the current node is the leftmost node then the cursor is moved to the previous sibling of the parent of the current node. If the current node is the leftmost node in the tree (as defined in pre-order) then the command has no effect.

Cursor-back

.BACK (^b)

Moves the cursor back to its previous position, provided that the last command was a cursor moving command.

Terminal

.TERMINAL (^x.)

Searches the tree for the next terminal or file node. The search is restricted to the current window.

Rterminal

.RTERMINAL (^x,)

Searches the tree for the previous terminal or file node. The search is restricted to the current window.

B.2. Searching Commands

The various find commands all follow the unparsing scheme either forward or backward. The search shows the visual order of the screen. The following are commands of searching:

.FIND<string> (^f)

Searches the tree for a matching variable name, constant name, operator synonym, or operator name. The search is restricted to the current window. If no string is given one is prompted for. If a carriage return is typed for the string prompt the string specified in the previous search is used.

Rfind (^l)

.RFIND<string> (^xb)

Searches the tree in reverse for a matching variable name, constant name, operator synonym, or operator name. The search is restricted to the current window. If no string is given one is prompted for. If a carriage return is typed for the string prompt the string specified in the previous search command is used.

Class

.CLASS<string> (^xn)

Searches the tree for the first node in the specified class. The search is restricted to the current window. If no string is given one is prompted for. If a carriage return is typed for the string prompt the string specified in the previous search command is used.

Rclass

.RCLASS<string> (^xp)

Searches the tree in reverse for a node in the specified class. The search is restricted to the current window. If no string is given one is prompted for. If a carriage return is typed for the string prompt the string specified in the previous search command is used.

First

.FIRST (^x b)

If the current node is on a list, then the cursor is moved to the first item on the list. Otherwise the cursor is moved to the first sibling.

Last

.LAST (^x^l)

If the current node is on a list, then the cursor is moved to the last item on the list. Otherwise the cursor is moved to the last sibling.

Numerical Argument for Cursor Movement

.<number>

The explicit cursor moving commands ("cursor-in", "cursor-out", cursor-next, cursor-previous) have an optional parameter that precedes them. The numerical argument indicates how many applications of the given command should be made. The argument is not a command so that it can not be used alone.

B.3. Help Information

Operator Help

.HELP (^x?)

If the current node is a meta node, .HELP displays the list of applicable language commands (and their synonyms). Otherwise, the list of editing commands is displayed.

Command help

.?

Displays the list of editing commands (and their synonyms).

B.4. Tree Manipulation

Clip Subtree

.CLIP<tree-name> (^k)

Clips current subtree into a named tree which is kept in the clipped area separate from the main tree. The name of the tree can be specified following the command or it will be prompted for.

Insert Subtree

.INSERT<tree-name> (^x^i)

Inserts a clipped subtree at the current node (which must be a meta node) provided that the root operator of the subtree is legal in this position. If no tree name is specified one is prompt for.

Move Subtree

.MOVEFROM (^xf)

Clips current subtree into the MOVE tree which is kept in the clipped area separate form the main tree. If cursor points to a member of a list, then a request will be given to place the cursor on the end of the list and press <CR>. The nodes clipped are deleted from the tree. The combination of .MOVFFROM and .MOVETO is the preferred way to move a segment of the tree to a new location.

.MOVETO (^xt)

Inserts current subtree in the MOVE tree at the current node (which must be a meta node or on a list) provided that the root operator of the subtree is legal at this position. The combination of .MOVEFROM and .MOVETO is the preferred way to move a segment of the tree to a new location.

Extend List

.EXTEND (^e)

Extend a list with a new meta node. If the current node is a list node (variable arity node) then an element is created at the beginning of the list. If the current node is a member of a list then the meta node is inserted immediately after it.

Prepend to List

.PREPEND (^x^a)

If the current node is a member of a list, it places a meta node at the beginning of the list.

Append to List

.APPEND (^x^e)

If the current node is a member of a list, it places a meta node at the end of the list.

Delete

.DELETE (^d)

Deletes the current subtree. If the subtree is an element of a fixed arity node, then a meta node is inserted in its place. If the subtree is an element of a list, the element is removed completely from the list.

Replace

.REPLACE (^r)

Deletes the current subtree. If the subtree is an element of a fixed arity node, then a meta node is inserted in its place. If the subtree is an element of a list, the element is replaced by a meta node of the appropriate class.

Nest

.NEST<operator name> (^x^n)

Takes the current subtree and nests it into a subtree that will have the operator as root operator. The operator name can be given following the command or it will be prompted for. A nesting that would result in an invalid tree is not permitted.

Unnest

.UNNEST (^xo)

Undoes the operation of the .NEST command. In most cases, it is not required since .DELETE does an unnest in the same circumstances. Thus if the user uses .NEST to transform "a" to "a+b" then either .DELETE applied to b or .UNNEST applied to "a" will create "a".

Transform

.TRANSFORM <operator name> (^t)

Transforms the operator of the current node to the desired one. For the transformation to succeed, the new operator must be in the same class as the old one and the respective offspring must also match exactly.

B.5. Move to Marked Node

A limited number of nodes can be marked for later display or return. An option exists while messages and errors are displayed to mark selected nodes. The first such element marked will be the new current node. Only ten nodes are currently saved.

Operator Set-mark

.SET-MARK (^xm)

Mark the node displayed. Additional assignments replace the entry following the cursor mark.

Operator Unset-mark

.UNSET-MARK (^xu)

Remove mark at the node displayed. Marks are also removed by replacement or deletion.

Operator Nmark

.NMARK (^xk)

Set the current mark to the next mark on list. The list is treated circularly.

Operator Pmark

.PMARK (^xj)

Set the current mark to the previous mark on list. The list is treated circularly.

Operator Exchangemark

.EXCHANGEMARK (^x^x)

Move to the current mark. The previous location becomes the current mark. The new location is no longer marked. Two .EXCHANGEMARK commands will return to the original position.

Operator Marks

.MARKS (^x^d)

Highlight the current marks. The option is given for each mark display to continue the listing, stay at that mark and make it the current mark, or to stop the listing and

return to the original position.

B.6. Input/Output

Read program

.READPROG<file-name> (^x^r)

Reads a tree from a file. Checks that the file contains a valid tree. Replaces the current tree with the new tree. Checks with the user if the current tree has not been saved. The file name can be given after the command or given to the ALOE prompt.

Load Tree

.LOADTREE<file-name> (^x^v)

Loads a tree from a file into a clipped area. A clipped window is assigned to it with the name of the window taken from the file name. The name of the file can be given after the command or given to the ALOE prompt.

Write Tree

.WRITE (^x^w)

Writes a tree into a file in tree form. The default is the file name given at invocation of ALOE.

Unparse into File

.UNPARSE<file-name> (^wu)

Unparses the tree into a text file. The file name can be given after the command or to the ALOE prompt. Useful for producing printouts. Note that this command differs from .WRITE only in the form of the written file.

B.7. Exit ALOE Quit and Save

.QUIT (^x^f)

Saves the current tree in a file in tree format and leaves AIOF. It uses the file name given at the invocation with the extension ".tr".

Cancel

.CANCEL (^c)

Leaves ALOE. If the tree has been changed since the last .WRITE command, the user is warned and given a chance to abort the command.

B.8. Display Manipulation

Display Tree

.DISPLAY (^l)

The screen is cleared and redisplayed. Useful when operating system messages or other such noise gets

displayed in the screen.

Window Down

.WDOWN (^w^n)

Scrolls the tree window down by one half of the window length.

Window Left

.WLEFT (^w^a)

Scrolls the tree window left by one third of the window width.

Window Right

.WRIGHT (^w^e)

Scrolls the tree window right by one third of the window width.

Window up

Scrolls the window up by one half of the window length.

Select Window

.WINDOW<window-name> (^w^w)

The selected window is displayed on the screen. The window name can be given after the command or to the ALOE prompt. The window must be a tree or a clipped window. If it is a tree window, the appropriate context switch takes place as if the cursor had been moved there explicitly.

Select Clipped Window

.CLIPWINDOW<window-name> (^w^c)

The selected clipped window is displayed on the screen. The window name can be given after the command or to the ALOE prompt. The window must be a clipped window.

B.9. Other Commands

Fork a UNIX Shell

.! (^x!)

Calls the UNIX shell from ALOE.

Edit

.EDIT (^x^t)

Invokes the text editor specified by the environment variable EDITOR, to edit a constant, long constant,

or text node. Upon return the screen is updated to incorporate the edited string.

Set Mode

.MODE (^x^m)

Sets the mode to the new value. ALOE first prompts for the name of the mode and then for the new value of the mode.

Set Unparsing Scheme

.SCHEME<scheme-number> (^xs)

This is a command to let the user change the current unparsing scheme. Takes as an argument the number of the unparsing scheme. The scheme number can be specified after the command or given to the ALOE prompt. If the argument is out of range (larger than the largest defined unparsing scheme) then scheme zero is used.

APPENDIX C

CONSTRUCTIVE COMMANDS

AND	Calls an "and" template.
ARRAYTYPE	Calls an array type template.
ASSIGN	Calls an assignment statement template.
BLOCK	Calls a declaration and statement part of the program.
BOOLEAN	Prints a keyword "boolean"
CASE	Calls a "case" template.
CASELABEL	Calls a case label template of variant record.
CASELEMENT	Calls a 'case label' statement template.
CCOM	Calls a template which contain a comment after each constant declaration.
CHAR	Prints a keyword "char"
COMCONST	Calls a template containing a comment template before a constant declaration.

COMMENT	Calls a comment template.
COMPOUNDSTAT	Calls a compound statement template.
COMPROCFUNC	Calls a template containing a comment before a procedure or a function.
COMPROG	Calls a template containing a comment template before a skeleton of a Pascal program.
COMSTAT	Calls a template of a comment and a statement.
COMSTATPART	Calls a template of a comment and a statement part of the program.
COMTYPE	Calls a template of a comment and a type declaration.
COMTY	calls a template containing a comment template after each type definition.
COMVAR	Calls a template containing a comment template before a variable declaration.
COMVRL	Calls a template containing a comment template after each variable definition.
CONSTDECL	Calls a constant declaration template.

CONSTDEF	Calls a constant definition template.
DIV	Calls a 'div' template.
DIVIDE	Calls a "divide" template.
DWFOR	Calls a 'for downto' template.
EMPTY	Calls an empty node. This can be done by just typing <CR>.
EOF	Calls an "eof" template.
EOLN	Calls an "eoln" template.
EQ	Calls an "equal" template.
EXP	Calls an expression template.
FIELDESIGN	Calls a record variable template.
FIELDID	Calls a field identifier template.
FIELDIDENT	Calls a field identifier of a record template.
FILEIDENT	Calls a file identifier template.
FILETYPE	Calls a file type template.
FIXPART	Calls a fix part of a record section.

FIXVARIANT	Calls a fix and variant part of a record section.
FORWARD	Calls a "forward" statement template.
FUNCT	Calls a function template.
FUNCTNOPARAM	Calls a function heading which has no parameter passing.
FUNCPARAM	Calls a function heading which has parameter passing.
FVAL	Input the first value of a width field in a write statement.
GOTO	Calls a 'goto' template.
GE	Calls a "greater than or equal" template.
GT	Calls a "greater than" template.
IDENT	Inputs an identifier. It is not necessary to type in this command before typing in an identifier name.
IF	Calls an "if" template.
IFELSE	Calls an "if then else" template.
IN	Calls an "in" template.

INDEXVAR	Calls an index variable template.
INPUTLIST	Calls a list of inputs in a 'readln' statement.
INTCONST	Inputs an integer constant.
INTEGER	Prints a keyword 'integer'
LABELPART	Calls a "label" declaration template.
LABELSTAT	Calls a 'label' statement template.
LE	Calls a "less than or equal" template.
LT	Calls a 'less than' template.
MINUS	Calls a "subtract" template.
MOD	Calls a 'mod' template.
MULT	Calls a "multiply" template.
NE	Calls a 'not equal' template.
NEGATE	Calls a "unary minus" template.
NIL	Prints a keyword 'nil'
NORANGESET	Calls a set-expression which is a set value specified as [list of elements].

NOT	Calls a "not" template.
OR	Calls an 'or' template.
OUTPUTLIST	Calls a list of output in a writeln statement.
PACKSTRUCT	Calls a packed structure template.
PARAMGROUP	Calls a parameter group template.
PGMHEAD	Calls a program heading template.
PLUS	Calls a "plus" template.
POINTERTYPE	Calls a pointer type template.
POINTERVAR	Calls a pointer variable template.
PROCCALL	Calls a "procedure call" template.
PROCCALLNOPAR	Calls a "procedure call" template which has no parameters.
PROCED	Calls a procedure template.
PROCFORMALHEAD	Calls a procedure heading which has a formal parameter list.
PROCHEAD	Calls a procedure heading which has no formal parameter list.

PROCPARAMS	Calls a formal parameter template which is a procedure.
PROFUNCS	Calls a procedure or function template.
PROG	invokes complete skeleton of PASCAL program.
RANGESET	Calls a set-expression which is a set value specified as [subrange].
READ	Calls a "read" template.
READLN	Calls a 'readln' template.
REAL	Prints a keyword "real"
REALCONST	Inputs a real constant.
RECORDSECTION	Calls a record section of a record type.
RECORDTYPE	Calls a record type template.
REPEAT	Calls a "repeat" template.
REWRITE	Calls a "rewrite" template.
RESET	Calls "reset" template.
SCALAR	Calls a scalar type template.
SET	Calls a set template.

SETTYPE	Calls a set type template.
SIMPLETYPE	Calls a simple type template.
STATPART	Calls a statement part of the program.
STRING	Calls a string type template.
SUBRANGE	Calls a subrange type template.
SVAL	Calls a second value of a width field in a write statement.
TEXT	Prints a keyword "text".
TFOR	Calls a 'for to' template.
TYPEDEF	Calls a type definition template.
TYPEDEFPART	Calls a type declaration template.
TYPEIDENT	Calls a type identifier template.
VARCOMPONENT	Calls a variable template.
VARDECL	Calls a variable definition template.
VARDECLS	Calls a variable declaration template.
VARIANT	Calls a case label statements.
VARIANTPART	Calls a variant part of a variant record.

VARPARAMGROUP	Calls a variable parameter template.
WHILE	Calls a 'while' template.
WIDTHFIELD	Calls an output template which has a width field in a write of writeln statement.
WITH	Calls a "with" template.
WRITE	Calls a "write" template.
WRITELN	Calls a "writeln" template.

APPENDIX D

TERMINAL NODE TYPES AND UNPARSING SCHEME COMMANDS

D.1. Terminal Node Types

There are a variety of terminal nodes types:

(1) Statics are nodes that represent some concrete piece of the language, like the name of a type.

(2) Character constants are nodes containing a single ASCII character.

(3) Strings are character constants nodes that contain blanks.

(4) Variable nodes are automatically entered into a simple name table structure.

(5) Integer nodes are constants that normally contain integer values.

(6) Real nodes are constants that normally contain real values.

(7) User nodes are identical to string constants, except that the lexical routine can be other than the (usually default) lexical routine for string constants. Multiple

user nodes can be defined, each with different lexical routines if desired. User nodes are typicelly used for comment fields.

(8) Userconstant nodes contain strings of non-blank characters. Multiple user nodes can be defined, each with different lexical routines if desired.

The following default lexical routines are used:

Static {s}	<none>
Character {c}	lexchar
String {a}	lexstring
Variable {v}	lexvariable
Integer {i}	lexinteger
Real {r}	lexreal
User {e}	<none>
Userconstant {u}	<none>

D.2. Unparsing Scheme Commands

Each non-terminal and terminal operation must have at least one unparsing scheme associated with it. Each unparsing scheme is a string which has descriptions of the text to be used, the syntactic sugar to be used, and the way the actual object is to be formatted. The formatting commands available in unparsing schemes are:

<n> number

@n Insert a new line in the output.

@t or @> Insert four spaces in the output.

@< Go back four characters (stopping at the beginning of the line).

@+ Increase the indentation level (to take effect at the next '@n'). Every indentation is four spaces.

@- Decrease the indentation level (to take effect at the next "@n").

@l Flush left (start a new piece of output at the left margin of the current line).

@h Go back one character (stopping at the beginning of the line).

@b Go back to previous line (undo '@n').

@u<n> Change the current unparsing scheme to <n>. Push the current scheme index on a one-level stack.

@u Reset unparsing scheme to value of the one-level stack.

@p<n> Push marker <n> onto stack. Markers are used to "remember" column positions for formatting. They are specially useful when the desired formatting depends on the size of identifiers. There are four markers (numbered zero through three).

@r<n> Pop marker <n>.

@g<n> Get marker <n>. Moves the unparsing cursor to the column position specified by the marker.

@@ Display an "@" character.

@% Display a "%" character.

The way the objects of the nodes are displayed is different for terminals and non-terminals. For terminals. the following unparsing commands are available:

@c Display value of character constant, string, text, integer, real, boolean, user node, or userblanks node.

@s Display variable name from symbol table.

The unparsing commands available for non-terminals are:

@<n> Unparse the <n>th offspring recursively. Used only for fixed arity nodes where the offspring are numbered from one on up. For example "while (@1)@+@n@2@-@n" specifies that the node should be unparsed starting with the string "while (" followed by a recursive invocation of the unparser on the first offspring, a ")", a line break. The order in which the offspring are unparsed can be different from the one specified in the abstract syntax. The "n" in "<n>" refers to the abstract syntax specification order. Finally, nodes can be hidden (made "non-visible") by simply omitting them in the unparsing scheme.

<pr>@Ø<t>[@q<po>][@e<s>]

Unparse the list node. Used only for non-terminals that are list nodes. The "@Ø" indicates that each element of the list should be unparsed in order. The "<pr>" is the prelude string that should be printed before the list is unparsed. The string "<t>" is used to separate list elements ("t" is terminated by either the following "@q" or the end of the unparsing scheme). The string "<po>" is the postfix that is printed after the list is unparsed. The optional "@e" indicates how the list should be unparsed if it is empty (i.e. has no

current elements.) All of the strings may contain text and other unparsing commands. The part in square brackets are optional. If no empty specification is given then nothing will be unparsed when the list is empty. For example, the scheme 'versions:@+@n@@;@n@q@-@nend@e<no version>' specifies that the list should be unparsed starting with the string 'versions:', a line break and then the elements of the list separated by a ";" and a new line. The list should be terminated by a new line and the word "end" aligned with "[versions:]. If the list is empty then it should only unparse the string 'no version'.

@x Used only for non-terminals that have filenodes associated with them and indicates that the subtree is not "visible".

@z Used in conjunction with '@x' to specify the place where the name of the filenode should be placed.

All the letters after the "@" in unparsing commands may be either upper or lower case. Additionally, anywhere an "@" can occur a "%" can also be used with one exception. The exception is that in fixed arity operators, '@<n>' means that the node should be unparsed and visited, while "%n"

means to unparse but not to visit the node. In the case of lists it means that no element of the list can be visited.