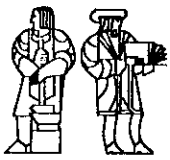
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

VLOE: A VAL Language-oriented Editor

Computation Structures Group Memo 255
September 1984

Steven C. Markowitz

Thesis submitted in partial fulfilment of the requirements for the S.B. degree of the Department of Electrical Engineering and Computer Science of the Massachusetts Institute of Technology.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

VLOE: A VAL Language-Oriented Editor

by

Steven C. Markowitz

Submitted to the
Department of Electrical Engineering and Computer Science
on September 4, 1984, in partial fulfillment of the requirements
for the Degree of Bachelor of Science

Abstract

Structure-Oriented program editors have been designed for several languages, in order to reduce or eliminate the edit-compile cycle of program development. These editors operate directly on parse tree representations of programs, rather than textual representations, and aid the programmer in detecting or avoiding syntax errors during program construction.

This thesis describes the design and capabilities of a strctured program editor for the language VAL. The VAL editor is based on the use of templates for common program constructs. For each type of VAL expression (e.g. LET-IN, IF-THEN-ELSE, FORALL-CONSTRUCT), the editor provides a 'template' which contains the keywords for that program construct and 'placeholders' for fields to be filled in by the user. These fields (e.g. conditional and consequent fields in an IF-THEN-ELSE expression) can be filled in with additional templates, entered via. editor commands, or with 'phrases' of text typed directly by the user. The editor automatically parses program phrases, and notifies the user of any syntax errors. This makes it possible to detect and correct errors, without having to compile the program being edited.

Table of Contents

Chapter One: Introduction 6

1.1 Edit-Compile Cycle 6
1.2 Semantic Trees 7

Chapter Two: Two Types of Structure-Oriented Editors 9

2.1 Free Form Editors 9
2.1.1 General Description 9
2.1.2 Advantages and Disadvantages of the Free-Form Approach 10
2.2 Template-Based Editors 11
2.2.1 Templates, Placeholders and Phrases 11
2.2.2 The Current Focus 13
2.2.3 Parsing 13
2.2.4 Advantages and Disadvantages of the Template-Oriented Approach 14

Chapter Three: Overview of VLOE 16

3.1 Selection of a General Approach 16
3.2 Construction of a Simple Program Using VLOE 17
3.3 Additional Editor Commands 22
3.3.1 Cursor Motion 23
3.3.2 Editing Phrases 23
3.3.3 Inserting and Deleting Templates 24
3.3.4 Opening Space 24
3.3.5 Comment Facility 25
3.3.6 Help Facility 25

Chapter Four: Design 26

4.1 The Main Modules 26
4.1.1 The Driver Loop 26
4.1.2 Help 26
4.1.3 Workspace 28
4.1.4 Window and Screen 28
4.1.5 Prog 29
4.1.6 Template 29
4.1.7 Field 31

4.1.8 Phrase 33
4.1.9 Parser and Scanner 33
4.1.10 Buffer, Line and Fragment 34
4.2 Semantic Checking 34
4.3 Interface to File System 34

Chapter Five: Implementation 35

5.1 Representing the Current Editor State 36
5.2 Screen Mapping Problem 38
5.3 File Implementation 41

Chapter Six: Evaluation 42

Chapter Seven: Possibilties for Future Work 43

References 44

Appendix A: Subset of VAL Handled by VLOE 45

Appendix B: Complete List of VLOE Commands 46

Table of Figures

Figure 1-1: Tree representation of plus1 program 8
Figure 2-1: Template for PL/I IF statement 12
Figure 3-1: Abs_plus_1 Program 18
Figure 3-2: Initial Configuration of VLOE 18
Figure 3-3: The cursor has been moved to the ID field. 18
Figure 3-4: The cursor is now pointing to the ID placeholder. 20
Figure 3-5: The first character of the function name has been typed. 20
Figure 3-6: The entire function name has been entered. 20
Figure 3-7: The function header has been completed. 21
Figure 3-8: The IF template has been inserted. 21
Figure 3-9: Almost finished. 21
Figure 4-1: Module Dependency Diagram for VLOE 27
Figure 5-1: Tree Representation of Absolute Value Program 37
Figure 5-2: Cursor stack configuration 39
Figure 5-3: Organization of absolute value program into fragments and lines 40

Chapter One

Introduction

1.1 Edit-Compile Cycle

One time-consuming task faced by programmers is the detection and removal of syntax errors. Typically, a programmer will write a program using some conventional text editor, compile it, and discover that syntax errors are present. In order to remove the error, the programmer must re-edit and recompile his program, and he may make new errors while trying to correct the old ones. Even after all syntax errors have been removed, new errors may be introduced as the programmer modifies his program to make it run correctly. This process of editing and recompiling a program several times has been termed the "edit-compile cycle" [2].

During the past few years, a tool has been developed to help programmers avoid the edit-compile cycle. This tool is the structure-oriented program editor. Unlike conventional text editors, a structure-oriented editor continually parses a user's program as it is being constructed, and notifies the programmer of syntax errors as soon as they are made. This makes it possible to correct syntax errors immediately, before the program is compiled. Normally, these editors are designed to deal with programs written in one specific programming language, although attempts at language-independence have been made [5, 7]. Ideally, this type of editor would eliminate the edit-compile cycle completely, allowing the programmer to write programs much more quickly. In practice, some syntactic or semantic errors might not be detected by the editor; in this case, the effect of the edit-compile cycle would be reduced, but not eliminated.

In this thesis, I will describe the design and implementation of a stucture-

oriented editor which I have been developing. This editor is designed to facilitate the construction and modification of programs written in VAL (Value-Oriented Algorithmic Language), a functional programming language developed by the Computation Structures research group at M.I.T.'s Laboratory for Computer Science. The VAL Language-Oriented Editor (VLOE) allows a user to construct and manipulate programs written in a subset of VAL, and could (in principle) be extended to handle the entire language.

1.2 Semantic Trees

Unlike conventional text editors, a structure-oriented editor does not operate directly on a text representation of a user's program. Instead, this type of editor maintains a parse tree of the program being constructed. Although the user's program is displayed on a terminal screen in the usual manner, editor commands are not defined in terms of lines or characters on the screen. Instead, the editor provides operations to make specified changes to a program's syntactic structure. Commands for cursor motion, and insertion and deletion of program elements are specified in terms of the program's semantic tree.

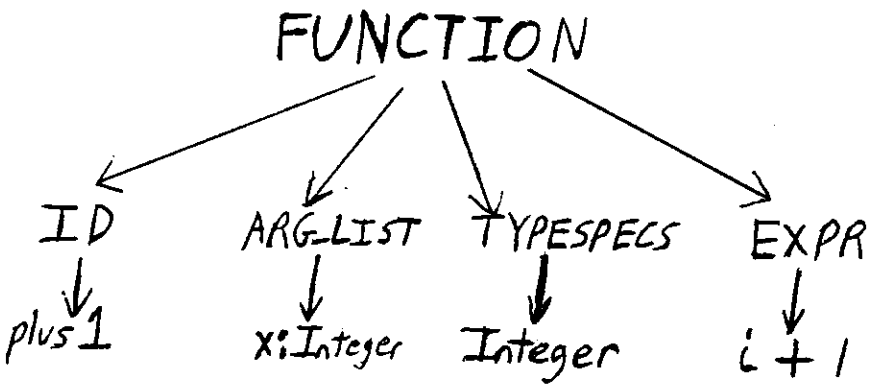
The tree structure of the following program is shown in Figure 1-1.

```
FUNCTION plus1 (i:Integer RETURNS Integer)
    i + 1
    ENDFUN
```

The program's semantic tree has a single FUNCTION node as its root. At next lower level, there are a series of nodes, one for each field of the FUNCTION: an ID (function name), ARG_LIST (argument list), TYPESPECS (return value types), and EXPR (the expression whose value will be returned). This tree describes the abstract syntax of the program; alternatively, a parse tree of the program's print

syntax could be used. The lowest level program constructs (e.g. the expression "i + 1") generally would not be stored in parse tree form. Moving DOWN from the root node brings us to the ID node, i.e. to the root's first child node. Moving to the NEXT node brings us to the ARG_LIST; moving from the ARG_LIST to the PREVIOUS node brings us back to the ID. Finally, moving UP from from the ID leaves us back at the root. These terms (UP, DOWN, NEXT and PREVIOUS element, HIGHER and LOWER level) will later be used to describe the operation of different editor commands.

Figure 1-1: Tree representation of plus1 program



Chapter Two

Two Types of Structure-Oriented Editors

2.1 Free Form Editors

2.1.1 General Description

The structure-oriented editors that have been designed can be divided into two general classes: free-form editors and template-oriented ones. One example of the former is the Pascal editor designed by Joseph Morris and Mayer Schwartz of the Applied Research Group at Tektronix [4]. This editor allows a user to construct and modify programs character by character, in the same manner as a conventional editor. The user's program is formatted by the editor according to its syntactic structure, and displayed in pretty-printed form at all times. The editor incrementally parses the user's program as each character is entered and as each modification to the program is made.

As soon as a syntax error is detected, the user is notified. When an error is detected, it does not have to be corrected immediately, but the editor's cursor is confined to the program region before the first error, until the error is corrected. If a new error is made before the first one is corrected, then the editor's cursor is confined to the (smaller) region before the new error. In this way, the editor guarantees that the program region from the beginning of the program up to the cursor is a viable prefix for a Pascal program.

Morris and Schwartz's Pascal editor maintains a set of several parse trees, each of which represents a region between discontinuities in program structure. (These dicontinuities may be caused by errors, or they may be present temporarily while a

program is being modified). When the user attempts to move the editor's cursor across the boundary between regions described by different parse trees, the editor attempts to rejoin the parse trees on either side of the border. If the error has been corrected, then the editor will be successful in rejoining the two trees, and the cursor can be moved past the location of the error which has been corrected.

2.1.2 Advantages and Disadvantages of the Free-Form Approach

The free-form approach to structure-oriented editing has two principal advantages. One of these is its similarity, from the user's point of view, to conventional text editing. Because programs are entered and modified character by character, in the same manner as with conventional editors, it may be less difficult for a new user to adjust to this type of editor than to an editor which is very different from conventional ones. Another advantage is the ease of modifying programs using a free-form editor. In order to change the first program fragment shown below into the second one, it is only necessary to delete the first and last lines, and the keyword THEN. This simple modification is more difficult when other types of structure-oriented editors are used. The features and advantages of free-form editors are more fully described in [4].

```
First fragment:

   IF x > 0
      THEN y := z + w
           a := b * c
           f := g / h
      END

Second fragment:

   y := z + w
   a := b * c
   f := g / h
```

Along with these advantages, however, the free-form approach has several drawbacks. A free-form editor can only notify the user of an error which has already been made; it cannot actually prevent the user from making an error in the first place. Typically, the editor's cursor is confined to the program region before the first syntax error; this prevents the user from examining, and correcting errors in, later portions of his program, until the earlier error has been corrected. In addition, the free-form editor must be capable of parsing the all of the language in which the user's program is written; this, combined with the need to be able to incrementally reparse the user's program and rejoin program subtrees could increase the size and complexity of the editor.

2.2 Template-Based Editors

2.2.1 Templates, Placeholders and Phrases

The other class of structured program editors can be termed "template-oriented". Some examples of this type of editor are the Cornell Program Synthesizer [6] for PL/I programs, ALBE/P (A Language-Based Editor for Pascal, developed at Yale) [3], and the Pascal Structure-Oriented Editor developed by Abraham Lederman at M.I.T. [2]. Instead of requiring the user to enter his program character by character, these editors provide a set of "templates" for common program constructs, which can be inserted by user command. Each template contains the keywords and punctuation for a particular kind of statement (e.g. IF-THEN-ELSE, WHILE-DO), as well as spaces called "placeholders", to be filled in by the user. For example, a template for an IF statement (would contain the keywords IF, THEN, ELSE, END, along with placeholders for the conditional,consequent and alternative clauses (See Figure 2-1). A single command would cause the entire template to be inserted into the user's program at the current

position of the editor's cursor. (If it would not be legal to insert the specified template at the current cursor position, the user would be notified, and the template would not be inserted.)

```
IF { Expression }
   THEN { Statement }
   ELSE { Statement }
   END;
```

Figure 2-1: Template for PL/I IF statement

These editors allow the user to insert additional templates into placeholders, or to type text directly into them. A line of text typed directly by the user is called a "phrase" [2, 6]. Only the simplest types of program constructs, such as arithmetic expressions, identifier names and assignment statements, can be entered as phrases; more complicated constructs must be built up from templates. It is possible for the user to line-edit the text within a phrase, by moving the cursor forward and backward inside the phrase, and by inserting and deleting individual characters. However, the user is not permitted to line-edit beyond either end of the phrase. For example, when editing the phrase "x > 0" in the program fragment "IF x > 0 THEN", it is not possible to delete individual characters in the keywords IF and THEN, or to move the cursor to characters in the middle of those keywords. (In fact, it is never possible to delete the individual characters within keywords; each keyword is a part of some template, which must be treated as a single unit).

The commands that template-oriented editors provide for cursor movement, program construction and program modification are based the tree structure of the program being edited. PSE, for example, provides commands to move the cursor up

or down one level in the program's semantic tree. However, PSE does not have commands to move up or down a specific number of lines.

2.2.2 The Current Focus

The operation of a template-oriented editor is based on the idea of the "current focus" [2]. The current focus is a subtree of the program being edited, to which the editor's cursor is pointing. For example, when the cursor on the screen is pointing to the "I" in the keyword IF, which is the first character in an IF statement, the current focus is the entire IF statement, not just the character "I". If a delete command was typed when the cursor was pointing in this position, the entire IF statement would be deleted. Whenever the cursor is pointing to the beginning of a template, that template is the current focus. If the cursor is inside a phrase, then that phrase is the current focus.

2.2.3 Parsing

Unlike the free-form editors, template-based editors do not parse a region of the user's program after each character is typed. Instead, these editors parse the current focus when an attempt is made to move the cursor away from it. If the current focus is a phrase, then it is parsed at this time, to verify that it is synactically correct and legal within the context of the enclosing template. For example, the conditional field of an IF statement must be a legal expression; the name field of a procedure must be a legal identifier name. If any syntax errors are present in the current focus, then the user is notified. The Cornell Synthesizer does not permit the user to move the cursor outside of the current focus until all of the errors inside it have been corrected.

Lederman's Pascal Structure-Oriented Editor also notifies the user of any syntax error within the current focus when the user tries to move away from it.

However, if the user immediately issues the command to move away a second time, then this second command will be obeyed. Thus, PSE notifies the user of errors, but allows him to move away from the current focus and correct the error later. This feature of PSE seems quite desireable. If the editor detects the use of a variable of the wrong type, for example, it is possible that the variable was actually used correctly, but not defined properly. PSE would permit the user to move directly to the erroneous variable definiton and correct it; the Cornell Synthesizer would force the user to change the current phrase (the expression containing the variable) before permitting him to move the cursor to the variable declaration. In addition, the user would have to move back to to the old current focus, to undo the changes that the editor forced him to make. For this reason, the approach use in PSE would appear to be preferable to the one taken by the Synthesizer.

2.2.4 Advantages and Disadvantages of the Template-Oriented Approach

The template-based approach to structure-oriented editing has several advantages. When this approach is used, the editor not only detects errors, but actually prevents certain errors them from being made. A template-based editor can make it impossible for the user to misspell keywords or to have missing or extra END statements in his program, because keywords and END statements are inserted by the editor as a part of templates. The use of templates should also enable the user to enter programs more rapidly, because the keywords and punctuation in templates do not have to be typed. A template-based editor would also provide some flexibility for the user, because the editor's cursor would not be restricted to the program region which precedes the first error (the free-form editor described in [4] imposes this restriction on the user).

There are two main disadvantages to the template-oriented approach. Although a template-based editor facilitates program construction, it can make

modification of an existing program more diffifcult. Consider the two program fragments shown below. Transforming one into the other is very simple if a free-form editor is used; the unwanted text can be deleted just as if a conventional editor was being used. In order to perform the modification using a template-oriented editor, however, each of the assignment statements in the THEN clause would have to be saved in a special buffer provided by the editor. The cursor would then be moved to the beginning of the IF-statement, and the whole statement would be deleted. Finally, each of the assignment statements would have to be retreived from the buffer and reinserted into the program.

```
First fragment:

   IF x > 0
      THEN y := z + w
           a := b + c
           f := g / h
      END

Second fragment:

   y := z + w
   a := b + c
   f := g / h
```

The other disadvantage of the template approach is its dissimilarity to conventional editors. In order to use a template-based editor, it is necessary to learn a set of commands which are quite different from those provided by conventional editors. It is also necessary to learn to think in terms of a program's tree structure, rather than its printed representation. Because the commands provided by a free-form editor more closely resemble those of a conventional editor, it may be easier for a new user to learn to operate an editor which is free-form than an editor that makes use of templates.

Chapter Three

Overview of VLOE

The editing systems described in the previous chapter provided the basis for my own work in designing and implementing the VAL Language-Oriented Editor (VLOE). VLOE is a structure-oriented editor designed to manipulate programs written in a subset of VAL (Value-Oriented Algorithmic Language, a functional programming language developed at M.I.T.'s Laboratory for Computer Science). The language subset handled by VLOE includes most common types of VAL expressions, including FUNCTION's, TAGCASE's, LET-IN, IF-THEN-ELSE, as well as a restricted form of the FORALL expression. A more detailed description of this subset can be found in Appendix A.

3.1 Selection of a General Approach

The first decision that had to be made in the design of VLOE was the selection of the general approach to be used: free-form or template-oriented. For several reasons, I decided that a template-based approach would be preferable. A template-based editor would not only detect syntax errors, but would actually prevent some of them from being made at all (e.g. misspelled keywords). Use of the template-based approach would also facilitate program entry by eliminating the need for the user to type the keywords and punctuation for each template. Because the purpose of the editor is to prevent errors and to reduce the time spent in editing and compiling, these would appear to be significant advantages.

A template-oriented editor should also use less computational resources than one that is free-form, because the latter would have to parse a portion of the user's

program after every character was typed. A free-form editor would require additional time to try to rejoin the parse trees of different regions of a program when the cursor was moved across the boundary between them (i.e. when an error was corrected, and the cursor was moved past its former location). The need to perform incremental parsing and to rejoin separate parse trees would also increase the size and complexity of the editor. Finally, if a template-based approach was used, the editor would not have to contain a parser for the entire VAL language; it would only be necessary to parse the simple expressions which could be found in phrases. This would further reduce the size of the editor.

3.2 Construction of a Simple Program Using VLOE

To provide an illustration of the operation of VLOE, I will show how the editor could be used to construct the program in Figure 3-1. This program consists of a single function, which computes the absolute value of a given real number, and adds one to the result. At the beginning of the editing session, VLOE constructs a single function template, with placeholders in all of its fields, to serve as a starting point for constructing the program.This function template is displayed on the screen of the user's terminal, as shown in Figure 3-2. The cursor on the screen (represented by an underscore in the figure) points to the beginning of the FUNCTION keyword.

The first step in constructing the program is to move the cursor DOWN into the tree structure of the function template. (The terms UP, DOWN etc. were defined in Section 1.2.) This is done by typing the DOWN command. (The word "DOWN" is not actually typed; commands are entered as control characters or escape sequences.) This command moves the cursor to the next lower level of the program's tree structure, i.e. to the first field of the function template. The screen now appears as in Figure 3-3. At this point, the current focus of the editor is the

```
FUNCTION abs_plus_1 ( x:Real RETURNS Real )
    IF x > 0
       THEN x
       ELSE -x
       ENDIF
    + 1
    % This line contains an irrelevant comment.
    ENDFUN
```

Figure 3-1:Abs_plus_1 Program

```
FUNCTION { ID } ( { ARG_LIST } RETURNS { TYPESPECS } )
    { EXPR }
    ENDFUN
```

Figure 3-2:Initial Configuration of VLOE

entire ID field, not any particular line within the field. This distinction becomes important when a field contains many lines. In order to notify the user that the current focus is an entire field, a '>' character is displayed at the current cursor position. This field marker is not a part of the user's program; it will be removed as soon as the cursor is moved to a new position.

```
FUNCTION>{ ID } ( { ARG_LIST } RETURNS { TYPESPECS } )
    { EXPR }
    ENDFUN
```

Figure 3-3:The cursor has been moved to the ID field.

Issuing the DOWN command a second time moves the cursor down one more level in the program structure, to the placeholder labelled { ID }. (VLOE displays placeholders as capitalized words enclosed in braces.) When the cursor is moved, the field marker disappears, and the screen appears as in Figure 3-4. The user can now fill in the placeholder with the name of the function being constructed. As soon as the first character of the function name is typed, the label for the placeholder disappears, and is replaced by the character that was typed. The remaining text on that line is shifted to the right, as in Figure 3-5, to leave a single blank space between the identifier name and the next character on the line. The remainder of the function name is then typed, and the text to the right of the cursor is automatically shifted one space to the right as each character is typed. The screen now appears as in Figure 3-6. If the characters in the function name were all deleted, by typing the delete command once for each character, the symbol for the placeholder would reappear, and the screen would appear as in Figure 3-4 again.

After the function name has been entered, the user moves the cursor to the ARG_LIST placeholder, to type the list of the function's formal parameters. To do this, the cursor is first moved UP one level in the program structure (via. the UP command), making the current focus be entire ID field. Next, the NEXT command is issued, the cursor is moved to the ARG_LIST field). Finally, the DOWN command is tyed, moving DOWN to the ARG_LIST placeholder. The list of function arguments could then be entered, by typing "x:Real". Similarly, the user could move the cursor to the TYPESPECS placeholder and type "Real", to enter the type of the return value. The screen now appears as in Figure 3-7.

In order to construct the IF expression, the cursor must first be moved to the EXPR field of the FUNCTION template, via. the UP and NEXT commands. Next, the INSERT_TEMPLATE command is typed. A message appears at the top of the screen, asking the user to indicate the type of template to be inserted. (This message

```
FUNCTION { ID } ( { ARG_LIST } RETURNS { TYPESPECS } )
    { EXPR }   cursor
    ENDFUN
```

Figure 3-4: The cursor is now pointing to the ID placeholder.

```
FUNCTION a( { ARG_LIST } RETURNS { TYPESPECS } )
    { EXPR }   cursor
    ENDFUN
```

Figure 3-5: The first character of the function name has been typed.

```
FUNCTION abs_plus_1( { ARG_LIST } RETURNS { TYPESPECS } )
    { EXPR }          cursor
    ENDFUN
```

Figure 3-6: The entire function name has been entered.

does not interfere with the display of the program text, because the top line of the screen is is only used for editor messages.) The user then types the keyword IF, followed by a carraige return. VLOE would verify that IF is a valid template type, which can be inserted at the current location, and the template would be inserted. The screen now appears as in Figure 3-8. The conditional, consequent and

```
FUNCTION abs_plus_1 ( x:Real RETURNS Real )
    { EXPR }
    ENDFUN
```

cursor

Figure 3-7: The function header has been completed.

```
FUNCTION abs_plus_1 ( x:Real RETURNS Real )
    IF { EXPR }
       THEN { EXPR }
       ELSE { EXPR }
       ENDIF
    ENDFUN
```

Figure 3-8: The IF template has been inserted.

```
FUNCTION abs_plus_1 ( x:Real RETURNS Real )
    IF x > 0
       THEN x
       ELSE -x
       ENDIF
    ENDFUN
```

cursor

Figure 3-9: Almost finished.

alternative clauses of the IF expression are filled in by moving the cursor to each of

these fields, and typing the phrases to be inserted. The result is shown in Figure 3-9.

After the IF expression has been constructed, it is necessary to open space for the two remaining lines of the program. In order to this, the cursor is moved to the beginning of the IF template, and the OPEN_AFTER command is issued. This command opens up space for a new line within the FUNCTION's EXPR field, immediately after the IF template, and moves the cursor to this line. The user types "+ 1" to fill in this line of the program, and issues the OPEN_AFTER command again. The editor opens a new line, immediately after the current one, and the comment can be inserted. The abs_plus_1 program is now complete.

3.3 Additional Editor Commands

VLOE provides many editing commands in addition to the ones mentioned in the previous section. Some of these functions are briefly described below. A list of all of the VLOE functions, and the specific control characters used to invoke them, can be found in Appendix B.

The functions performed by VLOE can be divided into several classes: cursor motion; line-editing phrases; inserting and deleting templates; interfacing to the file system; and providing assistance to the user. In order to make it less difficult for a new user to learn the VLOE functions, I have tried to make the command characters for them be the same as the characters which invoke the corresponding functions in the conventional text editor TED. (This was not always possible, because some VLOE functions do not correspond to any TED function).

3.3.1 Cursor Motion

VLOE provides commands to move the editor's cursor to different nodes in the tree structure of a VAL program. Cursor motion is specified in terms of a program's syntactic structure, not its representation on the screen. The basic cursor commands are UP, DOWN, NEXT and PREVIOUS. UP, DOWN and NEXT were described in Section 3.2; PREVIOUS moves the cursor to the preceding element in the current level of the program's tree structure. In addition to these basic operations, VLOE provides commands to move the cursor to the next (or previous) phrase, placeholder or syntax error in the program. These commands can cut across a program's tree structure. They move the cursor to the appropriate phrase, regardless of its level or position within the program tree. These commands can make it easier to place the cursor in a particular position, without having to first move up and then move back down to get there. These operations also provide a simple way of for the user to determine whether there are any syntax errors or placeholders remaining in a program.

3.3.2 Editing Phrases

VLOE provides a few commands which are needed for editing phrases. These operations are similar to those provided by conventional text editors. Two of these commands are FORWARD and BACKWARD, which move the cursor one character forward or backward one character within a phrase. VLOE also provides operations to delete the character immediately before or after the cursor. These commands cannot be used to move the cursor beyond the beginning or end of the current phrase, or to delete text outside the phrase.

3.3.3 Inserting and Deleting Templates

The main commands for inserting and deleting templates are INSERT_____TEMPLATE, DELETE_____CURRENT_____FOCUS, and FOCUS_TO_PLACEHOLDER. The INSERT_TEMPLATE operation has already been described. DELETE_CURRENT_FOCUS cuts the current focus out of the program, and stores it in a buffer. FOCUS_TO_PLACEHOLDER replaces the current focus with an appropriate placeholder, and stores the old current focus in the buffer. The program fragment saved in the buffer can later be reinserted at a new location, by using the PASTE command. (Note: the PASTE command has not been implemented yet.) The DELETE_CURRENT_FOCUS and PASTE commands are useful for rearranging different sections of programs. The program modification described in Section 3.2 could be carried out by using these commands.

3.3.4 Opening Space

There are two operations provided by VLOE to open space for new lines. These are the OPEN_AFTER command, mentioned in Section 3.2, and OPEN_BEFORE. OPEN_AFTER inserts an appropriate placeholder into the program being edited, at a point immediately after the current focus. OPEN-BEFORE inserts a placeholder immediately before the current focus. After the placeholder has been inserted, the cursor is automatically moved to it. These operations can be used to insert additional fields into a template. They would be used, for example, to add tag arms to a TAGCASE expression, or ELSEIF clauses to an IF expression.

The OPEN_AFTER and OPEN_BEFORE commands can also be used to open space for additional lines within a field. The new line appears either immediately before or immediately after the current focus, depending on which command is

used. If the OPEN_BEFORE command is used when the cursor is inside a phrase, then the phrase is split at the current cursor position, and the second half of it is placed on a new line of its own. Thus, when a phrase is being edited, OPEN_AFTER produces the effect that a carriage return would, if a conventional editor were used.

3.3.5 Comment Facility

VLOE allows the user to insert comments into programs, by associating each comment with a particular template. A comment can be attached to a template by moving the cursor to the beginning of the template, invoking the INSERT_TEMPLATE function, and typing the word "comment" instead of a template type. The comment associated with a template is displayed immediately above the rest of the template. Each line of the comment is preceded by a percent sign, which the convention used to denote comments in VAL. A comment is treated exactly like any other field of the template; the same commands that are used to move to ordinary fields and edit them are used to access and edit comments. In addition, a comment can be inserted within a phrase, by preceding it with a percent sign.

3.3.6 Help Facility

At the present time, VLOE provides only a limited help facility for its users. VLOE's HELP function displays a list of basic VLOE commands on the terminal screen, along with brief descriptions of each. A second function, CURSOR_HELP, provides a description of some additional commands that deal with moving the cursor and deleting sections of a program.

Chapter Four

Design

In order to make the construction, testing, debugging and modification of VLOE more manageable, I have attempted to design the editor in a fairly modular fashion. The editor's design calls for a main driver loop (VLOE), screen manager (Window), parser, and data structures representing the current editor state and the tree structure of a program being edited. The functions of each module are described below. The relationships between the different components of the editor are shown in the following module dependency diagram. (Figure 4-1).

4.1 The Main Modules

4.1.1 The Driver Loop

The top level module of the VAL Language-Oriented Editor is simply called VLOE. When the editor is run, this module prompts the user for the name of a file containing the program to be edited. (In order to construct an entirely new program, the user types a carriage return without specifying any file name). VLOE reads editor commands from the keyboard of the user's terminal, and calls the appropriate workspace procedures to execute them. VLOE is also responsible for the reading and writing of files.

4.1.2 Help

The help module is used to provide assistance to the user. It displays a list of VLOE commands on the terminal screen, along with descriptions of them. The help

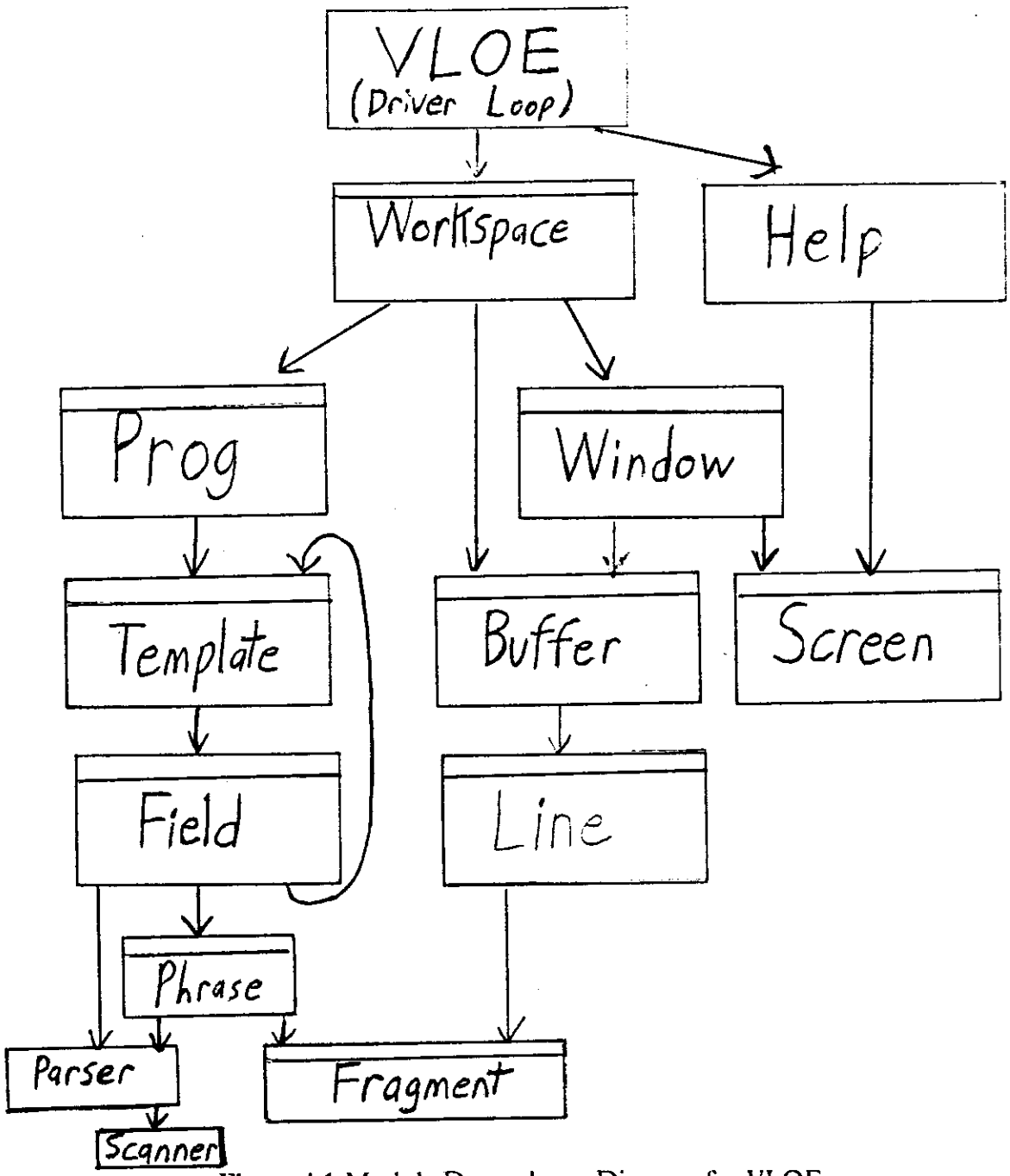


Figure 4-1: Module Dependency Diagram for VLOE

procedure operates directly on the screen object, to avoid unwanted interactions with the window. The screen can be repainted by VLOE (using the window's REPAINT operation) when the user is finished examining the information on the screen.

4.1.3 Workspace

The workspace is a data structure (or "cluster") containing most of the information describing the current editor state. It includes the semantic tree representation of the program being edited, a pointer to the current focus, a pointer to the specific character at which the cursor is located (if the cursor is inside a phrase), the cut buffer, and the window (screen manager). There is roughly a one-to-one correspondence between editor commands and workspace functions. Most editor commands can be executed by calling a single workspace function. In addition to functions for editor commands, the workspace provides a set of operations which are identical to the window functions of the same name; these operations simply call the appropriate window function. They are included in the workspace so that the driver loop will not need to have direct access to the window; this was done to maintain the modularity of the design.

4.1.4 Window and Screen

The window provides an interface between the workspace and the terminal screen. It is used to ensure that the appropriate portion of the program being edited is displayed on the screen at all times. The window provides operations to add characters to the screen, to erase characters, to insert or remove a group of lines from the screen, to repaint selected regions of the screen, and to display command arguments (e.g. filenames, template types) as they are being entered and line-edited.

The window performs its functions by calling the appropriate operations of

the screen module. The screen module is a simple cluster of operations which sends escape sequences to the terminal. The use of a separate screen cluster allows the window to ignore the details of different terminal types; the window specifies a simple screen operation (e.g. clear screen, scroll up) and the screen cluster transmits the character sequence for the type of terminal being used.

4.1.5 Prog

The prog is a cluster (data structure) which represents the top level of a program's semantic tree. A prog is essentially a list of the function templates at the top level of the structure of the program being edited. Prog operations are provided to add new function templates to the list, to cut out existing functions, to access the current function, and to obtain a pointer to the next or previous function template in the list. (The prog contains the actual function templates as its elements, not a list of function names).

4.1.6 Template

The template data type corresponds to the abstract concept of a template, on which VLOE and other template-oriented editors are based. Templates are used to represent common VAL program constructs, such as FUNCTIONs, LET-IN expressions, IF-THEN-ELSE, and FORALL-CONSTRUCT. A complete list of the templates used by VLOE can be found in Appendix A.

A template is essentially a list of *fields*, which correspond to items to be filled in by the user (e.g. conditional, consequent and alternative in an IF expression; decldefs and expression to evaluate in a LET-IN). The information associated with each field of a template can be accessed by the template and altered when necessary, as programs are constructed and modified. The following (incomplete) list of template operations provides an indication of the capabilities of the template cluster.

make_<template_type>
Constructs and returns a template of type <template_type>, with placeholders in all fields.

is_<template_type> Returns **true** if the template is of type <template_type>; otherwise, returns **false**.

get_current_field Each template has one field which is designated as the "current field". A template created using the **make** operation will have its first non-comment field as its current field. The **get_current_field** operation returns the current field of a given template.

get_next Constructs and returns a new template identical to the old one with the exeption of having a different field designated as "current field". The field after the current field of the old template will be the current field of the new template.

get_previous Similar to **get_next**, but the new current field is the field before the old current field, not the one after. Note that t.current___field = t.previous.next.current___field and t.current___field = t.next.previous.current___field, but t ~= t.previous.next.

open_after This operation inserts a new placeholder field into the template, at a position immediately before the current field. The new field is then becomes designated as the current field. This makes it possible for a template to have an indefinite number of fields; this feature is necessary because some templates (e.g. TAGCASE, FORALL) can have an arbitrary number of fields.

open_before Obvious

cut_field Removes the current field from the template, and returns that field. One of the adjacent fields will be designated as current.

clear_field Replaces the current field with a placeholder field of the same type.

insert_comment Adds a comment field to the template, if one is not already

present.

4.1.7 Field

A field is a portion of a template which must be filled in by the user after the template is created. A LET-IN template, for example, contains an optional comment field, one or more decldef fields (to be filled in with variable declarations and definitions), and an expression field to be filled in with an expression to be evaluated. A field consists of a list of *components*, each of which can be either a template or a phrase.

The designs of some other template-based editors do not include a separate data structure to represent fields. PSE, for example, requires every field of a template to be either another template or a single phrase. An editor for VAL, however, cannot impose this requirement. The syntax of VAL permits expression types represented by templates to be included in arithmetic expressions. For example, "2 * IF x=0 THEN 3 ELSE 4 ENDIF + 1" is a valid VAL expression, whose value is 7 when x=0 and 9 otherwise. This type of expression is permitted by some other languages, such as ALGOL 60, but not by Pascal of PL/I. An template-based editor for VAL (or ALGOL) would have to allow IF templates, as well as other expression templates, to be inserted into the middle of phrases.

In VLOE, the combination of phrases and templates in a single field is accomplished through the use of a separate data type representing fields. A field object consists of a tag identifying its type (e.g. expression, comment, identifier), and a list of components, each of which may be either a template or a phrase. Field operations are provided to access components, to insert and remove components, and to insert templates into components which are placeholders. With the exception of this last operation, most field functions are similar to the corresponding template

operations. The operations that templates provide to insert and remove fields are provided by the field cluster to insert and remove components. Some of the more important field functions are listed below.

make_placeholder Constructs and returns a field of the appropriate type, with a placeholder phrase as its only component.

get_current_component
Returns the current component of a field. (Each field has one component which is designated as the "current component". Fields created with the **make**operation have their first component designated as current. In this respect, fields are similar to templates.)

is_<field_type> Returns true if the field is of the specified type; otherwise, returns false.

get_previous, get_next
These operations return a new field derived from the old one by designating a different component as current. Their behavior is similar to the operation of the corresponding template functions.

insert_template If the current component of a function is a placeholder phrase, then this operation will insert a given template into that phrase, if it is legal to do so. (Actually, the current component, which is a phrase, is removed from the field and replaced by a new one which contains the template.

clear_component Changes the current component into a placeholder phrase. Inverse operation to insert template. Similar to the corresponding template operation.

cut_component, open_before, open_after
Similar to the corresponding template operations.

4.1.8 Phrase

A phrase is a piece of program text typed directly by the user, occupying all or part of a single line. Phrases may not contain templates or other subcomponents. Each phrase contains a fragment of text typed by the user, as well as a *default string.* The default string is the text to be displayed if the phrase is a placeholder. If the phrase does not contain any text (i.e. it is a placeholder), then this default string can be displayed on the terminal screen, to indicate the presence of a placeholder phrase of the appropriate type. The fragments of text enclosed in braces in previous illustrations (Figures 1-1 and 2-1) are examples of default strings.

The phrase data type provides operations to create a placeholder phrase with a specified default string, to insert or delete characters from the phrase, and to split a phrase into two separate ones. These operations are used for line-editing of phrases. Other phrase functions are provided to return the text of a phrase, and to change the a phrase's default string.

4.1.9 Parser and Scanner

The parser and scanner are responsible for parsing the programs constructed by the editor. It is not necessary for the parser to be able to handle the entire VAL language; only phrases and simple field types need to be parsed. The scanner reads the text of the field or phrase being parsed and converts it into tokens, which can be passed to the parser; the parser can then determine whether or not the text is syntactically correct. If the text is not syntactically correct, or is not legal in its context, then the user is notified of the error, so that it can be corrected. (Note: the parser and scanner modules for VLOE have not been implemented yet.)

4.1.10 Buffer, Line and Fragment

The buffer, line and fragment clusters are used to organize the fragments of text stored in the program tree into a form which can conveniently be displayed by the window. Essentially, a fragment is a string of text which makes up a keyword or a phrase; a line is a list of fragments which should be displayed on the same line of the terminal screen; the buffer is a data structure which links together all of the lines in a program. Because the sole purpose of these clusters is to facilitate the mapping of the program tree structure onto the terminal screen, these data structures will be described in Section 5.2, which deals with screen mapping issues.

4.2 Semantic Checking

At the present time, VLOE does provide any facilities for semantic checking. Although there are several types of semantic checking which it could be useful for VLOE to perform, I believe that including semantic checking would be beyond the scope of this thesis.

4.3 Interface to File System

VLOE provides a set of three commands to interface to the file system: WRITE, WRITE_TEXT and READ. WRITE and WRITE_TEXT make a copy of the program being edited and write it to a specified file. The WRITE command produces a normal text copy, which can be printed or compiled; WRITE_TEXT produces a reprentation of a program's tree structure, in a form which can be read back by the editor. The READ command reads a file which has been constructed using the WRITE operation. The tree structure of the program stored in this file is reconstructed by the VLOE, and the user can then continue to edit the file.

Chapter Five

Implementation

The VAL Language-Oriented Editor was implemented in CLU. This programming language provides powerful mechanisms for creating abstract data types (CLUSTERS); this facilitates the construction of highly modular programs. Each of the modules of VLOE shown in Figure 4-1 was implemented as a single CLUSTER, with the exception of the driver loop and help functions, which were implemented as procedures.

At the time of writing (September 4, 1984), a substantial part of the editor has been implemented. Commands for cursor motion, inserting and deleting templates, editing phrases, and reading and writing files all appear to work correctly. However, the PASTE command has not been implemented yet, and the parser and scanner have not been written. These functions will probably be added to the editor during the fall.

In the remainder of this chapter, I will describe a few aspects of the VLOE implementation that might not be evident from the design. In particular, I will discuss the manner in which VLOE represents progfram trees and keeps track of the cursor position within them, the implementation of the file system interface, and the way in which changes to the program tree are reflected on the terminal screen.

5.1 Representing the Current Editor State

There are two main components of the editor state that need to be represented. These are the tree structure of the program being edited and the position of the editor's cursor within this program. The program tree and the cursor information are both stored in the workspace module of the editor.

In order to represent the tree structure of programs, VLOE uses a hierarchy of data abstractions. The abstractions used are the prog, template, field and phrase clusters, which were described in the previous chapter. Each prog, template and field object contains a pointer to a linked list whose elements are the object's components. The prog, for example, contains a pointer to a linked list of function templates. Each template contains a pointer to a linked list of its fields; every field has a pointer to a list of its components. In addition, templates may contain additional pointers; these are used to point to any lines of the template which do not contain fields. The most common example of this is the END line at the end of many templates.

Unlike progs, templates and fields, phrases do not have a linked list structure. Each phrase contains only a string of text typed by the user. (Actually, phrases do not directly contain strings of text. Each phrase has a pointer to an object called a "fragment" which contains the phrase's text. This will be explained in Section 5.2.) The prog, template, field and phrase data types together make up the program tree. A short program and its tree representation are shown in Figure 5-1.

In addition to the program tree, the editor's workspace contains a representation of the current cursor position. The cursor position is represented by a stack of pointers, which is called the "cursor stack". The top element of the cursor stack points directly to the current position of the cursor within the program tree (i.e. to the current focus). The other elements of the stack point to the program

```
FUNCTION abs ( x:Real RETURNS Real)
    IF x > 0
       THEN x
       ELSE -x
       ENDIF
    ENDFUN
```

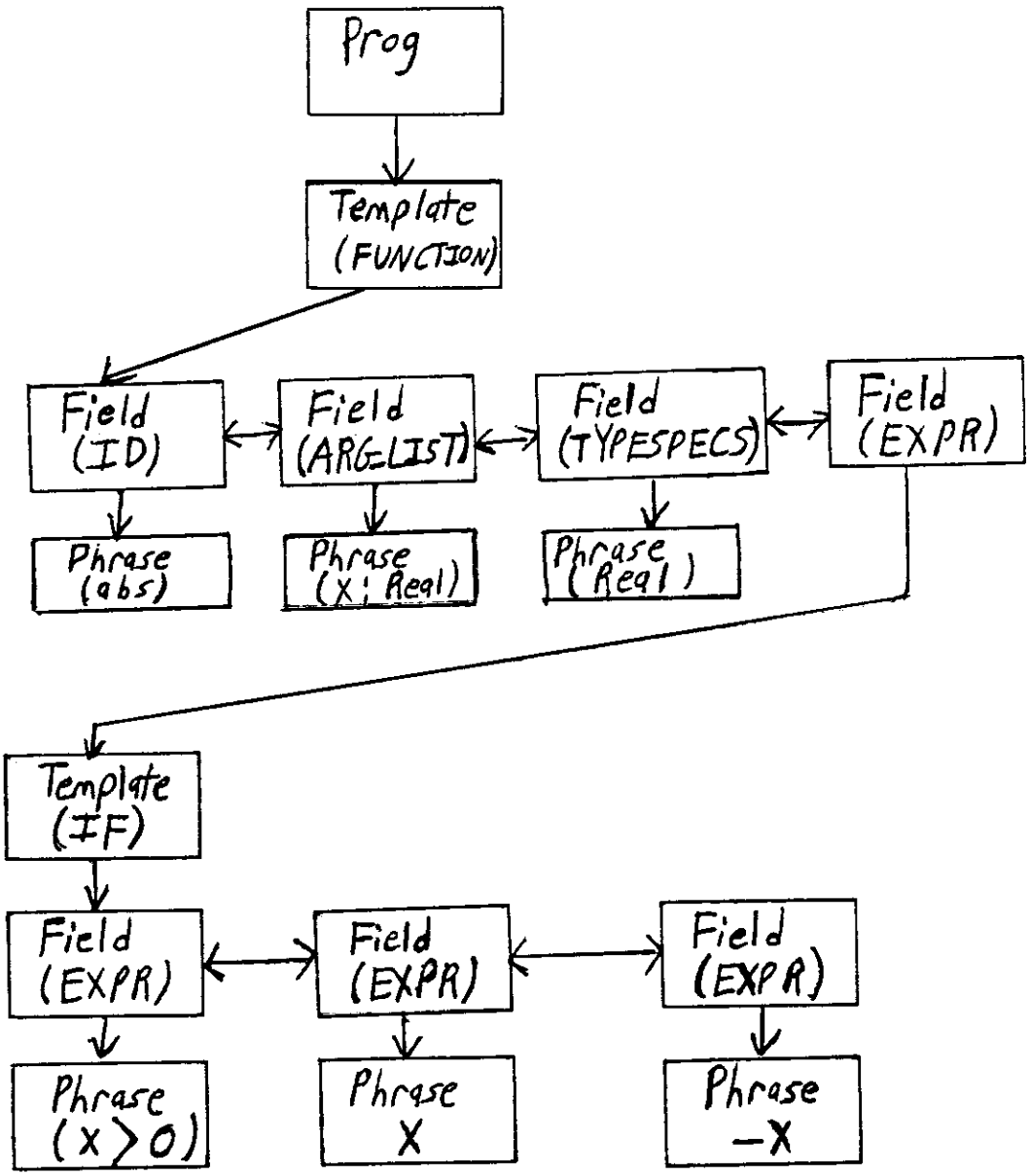


Figure 5-1: Tree Representation of Absolute Value Program

constructs surrounding the current focus. This organization of the cursor stack makes it possible to move UP one level in the program tree merely by popping the stack. To move DOWN, an additional pointer would be pushed onto the stack. Moving to the NEXT or PREVIOUS program element at the current level involves popping one pointer off of the stack and pushing a new one on. If the cursor is moved to a phrase, then an extra element is pushed onto the stack, to indicate which character of the phrase the cursor is pointing to. When the cursor is moved away from the phrase, this extra element is popped off of the stack. Figure 5-2 shows the configuration of the cursor stack, when the cursor is positioned at the last phrase of the program in Figure 5-1.

5.2 Screen Mapping Problem

One of the requirements for VLOE was that the editor should be screen-oriented. Every time the cursor is moved, and every time that text is inserted or deleted, the change should be reflected immediately on the screen. In order to do this, it is necessary to have a convenient way of mapping from the tree representation of a program to its positions on the terminal screen.

There are two basic approaches which can be taken to deal with this problem. One of these is to search through the program tree each time a change is made, to determine what should be displayed on the screen. The other solution is to maintain a representation of the program which is based on lines of text, rather than tree structure. VLOE takes the second approach to solving this problem. Instead of trying to translate from the tree structure to lines on the screen every time a change is made, VLOE maintains a set of data structures which organize the program into lines. These data structures are the buffer, line, and fragment data types, which were mentioned earlier.

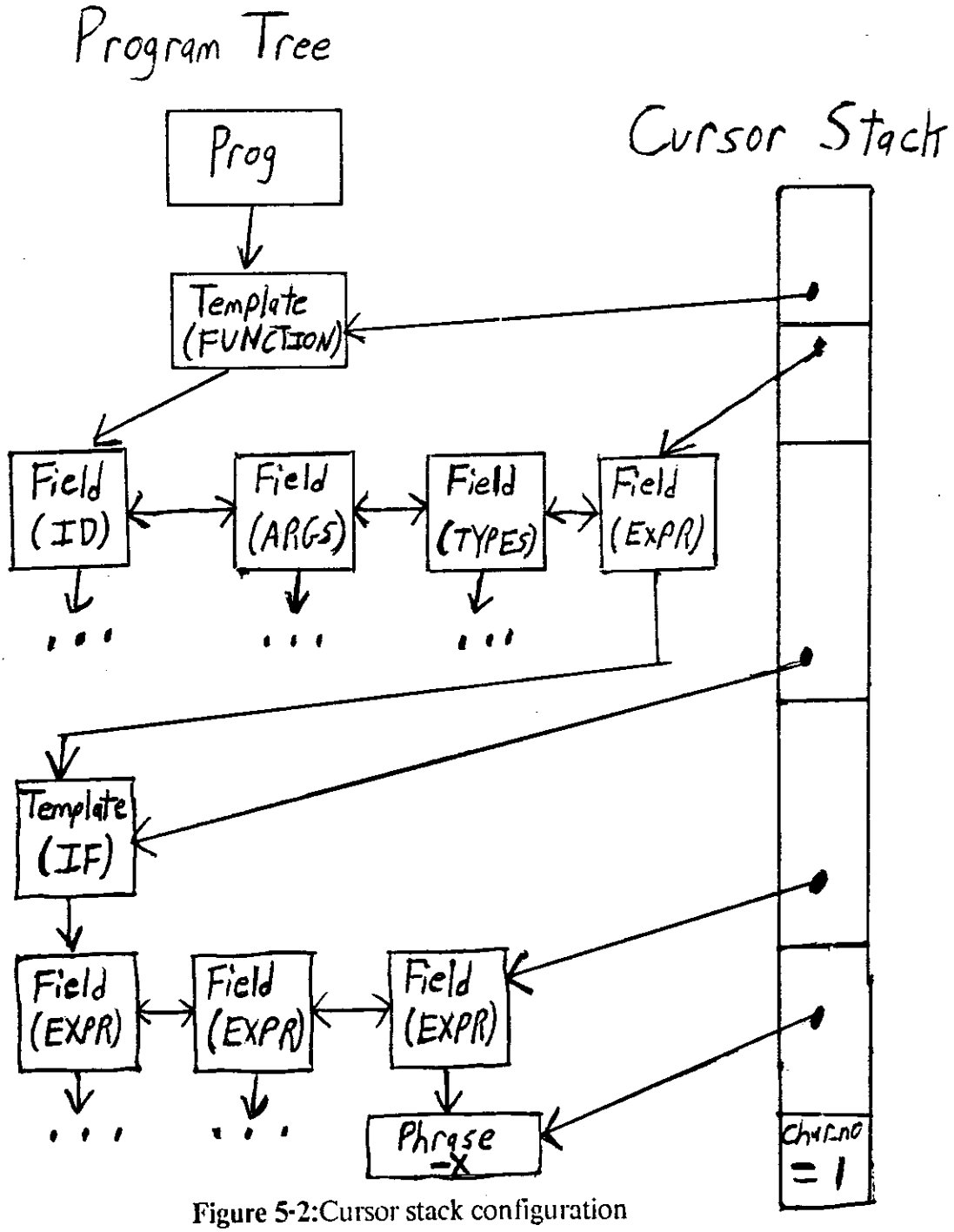


Figure 5-2:Cursor stack configuration

A *fragment* is a string of characters which should always be grouped together. A phrase, for example, keeps all of its text together in a single fragment. Each phrase object contains a pointer to the fragment associated with it. A phrase does not contain a separate copy of its text; in order to access its text, the phrase obtains it from the fragment.

Fragments of text which should appear on the same line of the screen are linked together into *lines*. The line cluster provides an operation which returns a single string containing all of the text in a line. This operation simply concatenates the pieces of text stored in the line's fragments. All of the lines in the program are linked together in a single *buffer* object. The organization of a program into lines and fragments is illustrated in Figure 5-3.

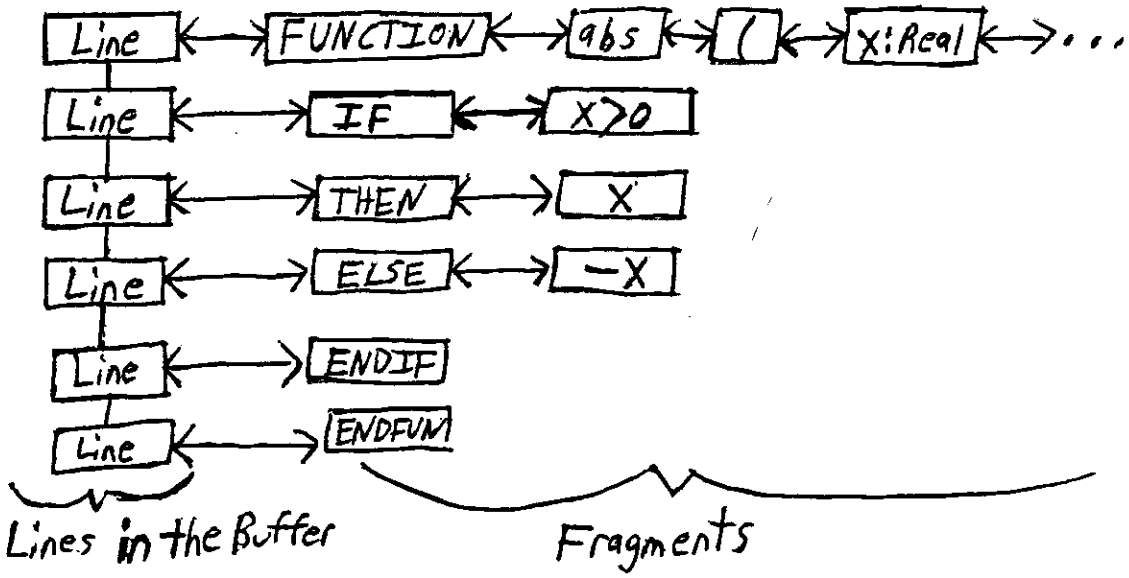


Figure 5-3:Organization of absolute value program into fragments and lines

When changes are made to the program tree, the appropriate changes are also made to the affected lines in the buffer. The operations for updating the screen can then be specified in terms of lines in the text buffer, instead of elements of the

program tree. The screen can be updated through the use of procedures that repaint specific lines on the screen with text from the buffer. As a result, the screen management cluster (window) does not need to know about program trees, or even about fragments. The screen manager for a conventional text editor would be able to perform all of the necessary functions. This simplified the construction of the VLOE window.

5.3 File Implementation

Implementation of the file system interface for VLOE was fairly straightforward. In order to write text files (when the WRITE_TEXT command is invoked), VLOE simply outputs each line in the text buffer. This means that writing text files is no more difficult for VLOE than for a conventional editor.

Because VLOE does not have the capacity to read ordinary text files, it was necessary for the editor to be able to write files in some other form, which it could read back. These files are written by using the WRITE command. The file representation of a program produced by VLOE is a string of characters which, if typed from the terminal keyboard, would cause the editor to construct the program again. A file produced in this way is approximately the same length as an ordinary text file for the same program. In order to read a file, it is only necessary for VLOE to disconnect its input stream from the terminal keyboard, and attach it to a stream from the file. VLOE executes the commands in the file as though they were typed from the keyboard. (VLOE does not update the screen until the entire file has been read and executed; this eliinates unwanted terminal output.) When the end of the file is reached, the input stream is reconnected to the keyboard, and the user can begin to edit the program.

Chapter Six

Evaluation

Because the parser for VLOE has not been written yet, I have not been able to measure the effectiveness of the editor in helping a programmer avoid syntax errors. However, I have obtained some idea of the performance of VLOE, by using it to construct several programs, and then constructing the same programs with a conventional editor. I have found the time needed to construct a program using VLOE to be comparable to the time needed when a conventional editor is used.

One problem that I have found in using VLOE is the difficulty of adjusting to using a structure-oriented editor for the first time. It was necessary to learn to think in terms of programs' tree structures, rather than lines and characters on the screen. It was difficult, at first, to determine which commands should be used to move the cursor to some desired location. I found it particularly confusing that the UP command, for example, could cause the cursor to move in any of three physical directions on the screen (up, left, and down), while moving UP in the program structure. This problem was alleviated, to some degree, by the addition of commands which cut across the tree structure of a program (e.g. next phrase, next placeholder). Hopefully, continued usage of VLOE, or experience with other structure-oriented editors, would further reduce these difficulties.

Chapter Seven

Possibilties for Future Work

The are a number of areas for future work in the development of VLOE. First, the implementation of the existing design could be completed. The PASTE function could be made operational, and the parser could be written. Implementation of the parser would involve writing the grammars describing each type of field or phrase that VLOE might have to parse. Separate grammars would describe the phrases which could legally appear at the beginning, middle or end of different kinds of fields.

Some other useful changes to VLOE would be the inclusion of semantic checking and the extension of VLOE to handle the entire VLOE language. It would also be desireable to enable VLOE to read ordinary text files and reconstruct the structure of programs from them. This would make it possible to use VLOE to edit programs that have already been constructed with conventional text editors.

References

[1] Ackerman, William B. and Dennis, Jack B.
VAL -- A Value-Oriented Algorithmic Language: Preliminary Reference Manual.
Technical Report 218, M.I.T. Laboratory for Computer Science, June, 1979.

[2] Lederman, Abraham.
A Pascal Structure-Oriented Editor: Design and Implementation Issues.
Master's thesis, Massachusetts Institute of Technology, 1981.

[3] Lewis, J. W. and Porges, D. F.
ALBE/P: A Languaged-Based Editor for Pascal.
In *Procedings of the Eighth Texas Conference on Computing Systems*, pages 5A-15 to 5A-19. IEEE, 1979.

[4] Morris, Joseph M. and Schwartz, Mayer D.
The design of a Language-Directed Editor for Block Structured Languages.
SIGPLAN Notices 16(6):28-33, June, 1981.

[5] Stromfors, O. and Jonesjo, L.
The Implementation and Experiences of a Structure-Oriented Text Editor.
SIGPLAN Notices 16(6):22-27, June, 1981.

[6] Teitelbaum, T. et. al.
The Why and Wherefore of the Cornell Program Synthesizer.
SIGPLAN Notices 16(6):8-16, June, 1981.

[7] Wood, Steven R.
Z - The 95% Program Editor.
SIGPLAN Notices 16(6):1-7, June, 1981.

Appendix A

Subset of VAL Handled by VLOE

At the present time, VLOE can only edit programs written in a subset of VAL. This subset includes the following types of VAL expressions: FUNCTIONS, IF-THEN-ELSE expressions (these expressions can contain ELSEIF clauses), TAGCASE expressions (ordinary tag arms and OTHERWISE arms are both allowed) and LET-IN expressions. A restricted form of the FORALL expression is also allowed. The FORALL expression may only contain one variable in its header; FORALL's involving more than one loop variable must be implemented by nesting two or more of FORALL expressions. The editor does not permit a FORALL to have a decldef-part. In addition to these restrictions, VLOE does not yet accept the TYPE, EXTERNAL or FOR-ITER constructs.

For a detailed description of the VAL language, see the VAL Preliminary Reference Manual [1].

Appendix B

Complete List of VLOE Commands

The section contains a brief description of each VLOE command, along with the specific control characters needed to invoke it:

Command:	Character to invoke command:	Description:
DOWN:	↑@, break, or ESC-B (Down arrow);	Move down one level in the program structure.
FIRST:	↑A	Move to the first character of the current phrase.
BACKWARD:	↑B	Move the cursor back one character within a phrase.
DELETE_CHAR:	↑D	Delete the character at the current cursor position.
END:	↑E	Move to the end of the current phrase.
FORWARD:	↑F	Move the cursor forward one character within a phrase.
QUIT:	↑G	Quit from typing a command.
HELP:	↑H	Print this help file.
OPEN_BEFORE:	↑J	Open space before the current focus.
REPAINT:	↑L	Clear the screen and redisplay.
OPEN_AFTER:	↑M	Open space after the current focus; split a phrase at the cursor.
NEXT:	↑N, or ESC-C (Right arrow);	Move to next program element at current level
PREVIOUS:	↑P, or ESC-D (Left arrow);	Move to previous program element at current level.
READ:	↑R	Read file.
UP:	↑U, or ESC-A (Up arrow);	Move up one level in the program structure.
Text/Cursor Move:	↑V	Used to precede cursor commands. A list of these commands is on the next page.
WRITE:	↑W	Write VLOE file, which can be used for further editing.
EXIT:	↑X↑Z	Exit from VLOE.
INSERT_TEMPLATE:	↑Y	Insert template or comment.
WRITE_TEXT:	↑\	Write a text (human-readable) copy of the program.
DELETE:	delete key	Delete the character before the cursor.

Text/Cursor Commands (must be preceded by ↑V):

Command character:	Command description:
↑A	Move to the first program element at the current level.
↑B	Move to the previous placeholder.
↑D	Change the current focus into a placeholder.
↑E	Move to the last program element at the current level.
↑F	Move to the next placeholder.
↑G	Quit from command.
↑K	Delete the current focus.
↑N	Move to the next phrase.
↑P	Move to the previous phrase.
T	Move to the top of the program tree. (This command is T, not ↑T)
↑U	Change template to non-flat form
↑V	Change template to flat form
(The ↑U and ↑V cursor commands only apply to IF, FUNCTION and FORALL templates)	
?	Print this help file.