HP DECset for OpenVMS

Guide to Language-Sensitive Editor

Order Number: AA–PQ9LD–TE

July 2005

This guide describes the HP Language-Sensitive Editor (LSE) and explains how to get started using its basic features.

Revision/Update information:	This is a revised manual.
Operating System Version:	OpenVMS I64 Version 8.2
	OpenVMS Alpha Version 7.3–2 or 8.2
	OpenVMS VAX Version 7.3
Windowing System Version:	DECwindows Motif for OpenVMS I64 Version 1.5
	DECwindows Motif for OpenVMS Alpha Version 1.3–1 or 1.5
	DECwindows Motif for OpenVMS VAX Version 1.2–6
Software Version:	HP DECset Version 12.7 for OpenVMS
	HP Language-Sensitive Editor for OpenVMS Version 5.0

Hewlett-Packard Company
Palo Alto, California

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

Figures

Tables

Preface

This guide explains how to use the Language-Sensitive Editor (LSE) in the OpenVMS HP DECwindows environment. The commands described in this guide are Portable commands available for use on multiple platforms. However, LSE remains compatible with prior versions of LSE. Users can alternate between the OpenVMS and Portable command languages by using the SET COMMAND LANGUAGE command.

Most examples in this guide describe the DECwindows method of using LSE, but some command-line information is also provided for reference. For more command-line details, see the *DIGITAL Language-Sensitive Editor Command-Line Interface and Callable Routines Reference Manual* and the *HP DECset for OpenVMS Language-Sensitive Editor/ Source Code Analyzer Reference Manual*.

Intended Audience

This guide is intended for software engineers who are responsible for designing and writing source modules. However, anyone can use LSE to create and edit text files.

Document Structure

This guide contains the following chapters:

- Chapter 1 is an overview of LSE.
- Chapter 2 describes how to create text files or programming language code files.
- Chapter 3 describes how to perform language-related editing and writing tasks.
- Chapter 4 describes how to define your own templates and how to create and use environment files. It also includes two examples of environment files created with LSE.

Associated Documents

The following documents might also be helpful when using LSE:

- *HP DECset for OpenVMS Installation Guide* contains installation instructions for all HP DECset tools, including the LSE component.
- *HP DECset for OpenVMS Language-Sensitive Editor/ Source Code Analyzer Reference Manual* contains command dictionary, parameter glossary, command summary and translation table information.
- *DIGITAL Language-Sensitive Editor Command-Line Interface and Callable Routines Reference Manual* contains command-line interface information and OpenVMS-specific information.

References to Other Products

Some older products that HP DECset components previously worked with might no longer be available or supported by HP. Any reference in this manual to such products does not imply actual support, or that recent interoperability testing has been conducted with these products.

Note

These references serve only to provide examples to those who continue to use these products with HP DECset.

Refer to the Software Product Description for a current list of the products that the HP DECset components are warranted to interact with and support.

Conventions

Table 1 lists the conventions used in this guide.

Table 1 Conventions Used in this Guide

Convention	Description
$	A dollar sign ($) represents the OpenVMS DCL system prompt.
Return	In interactive examples, a label enclosed in a box indicates that you press a key on the terminal, for example, Return.
Ctrl/*x*	The key combination Ctrl/*x* indicates that you must press the key labeled Ctrl while you simultaneously press another key, for example, Ctrl/Y or Ctrl/Z.
KP*n*	The phrase KP*n* indicates that you must press the key labeled with the number or character *n* on the numeric keypad, for example, KP3 or KP-.
file-spec, ...	A horizontal ellipsis following a parameter, option, or value in syntax descriptions indicates additional parameters, options, or values you can enter.
. . .	A horizontal ellipsis in a figure or example indicates that not all of the statements are shown.
. . .	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being described.
boldface text	Boldface text represents the introduction of a new term.
`monospace boldface text`	Boldface, monospace text represents user input in interactive examples.
italic text	Italic text represents book titles, parameters, arguments, and information that can vary in system messages (for example, Internal error *number*).
UPPERCASE	Uppercase indicates the name of a command, routine, file, file protection code, or the abbreviation of a system privilege.
lowercase	Lowercase in examples indicates that you are to substitute a word or value of your choice.

(continued on next page)

Table 1 (Cont.) Conventions Used in this Guide

Convention	Description
mouse	The term **mouse** refers to any pointing device, such as a mouse, puck, or stylus.
MB1,MB2,MB3	MB1 indicates the left mouse button, MB2 indicates the middle mouse button, and MB3 indicates the right mouse button.

1 Introduction to LSE

This chapter introduces you to the HP Language-Sensitive Editor (LSE) and describes the following topics:

- An overview of LSE features
- Getting started
  - Understanding LSE concepts
  - LSE window task areas and menus
  - Issuing commands
  - Using command languages
  - Getting help

LSE is a multilanguage, advanced text editor that uses the HP Text Processing Utility (HP DECTPU) as underlying technology. With LSE, you can perform the following tasks:

- Compile programming code without exiting from the text editor.
- Review diagnostic information from compilations.
- Format language syntax constructs automatically.
- Use templates for language constructs.
- Outline programming code.
- Enter pseudocode for code design preparation.
- Extract documentation from programming code comments.
- Obtain online help about a programming language.

LSE enables you to customize your editing environment to your programming preference or style. You can also extend your editing environment to handle highly specialized editing needs through HP DECTPU.

1.1 LSE Features

LSE provides the following features:

- Design support

  You can write **pseudocode** that describes a program's structure before you write the actual programming language syntax. LSE provides a mechanism to preserve your design information in comments when you enter the source code.

  LSE has features for editing comments, including word wrapping within block comments, paragraph fill, and line-comment alignment.

  LSE also lets you view your program at various levels of detail by collapsing lines of code to hide detail, and expanding the resulting overviews to see more detail.

- Language-specific templates

  LSE contains **templates** for programming languages that provide quick and efficient methods of writing language-specific constructs. These templates assist in the design and implementation of source code. You can modify existing templates or define your own templates.

- System services and run-time library templates

  LSE provides templates for subroutine libraries. In addition, LSE allows you to define templates for **packages** of subroutine libraries.

- Error correction and review

  You can compile your source code from an LSE session. Supported compilers provide LSE with compilation diagnostics so you can review compilation errors in one editing window while displaying the related source in another window.

  In addition, LSE provides a mechanism for interfacing many language processors to the diagnostic-review facility. See the Software Product Description (SPD) for a complete list of supported languages.

- LSE customization

  You can customize your editing environment in LSE, which uses HP DECTPU as the base for its text-editing functions. HP DECTPU features include a compiler, an interpreter, and procedures for screen management and text manipulation. The HP DECTPU language is block-structured and provides loop definitions, conditional definitions, case definitions, and assignment statements, as well as many built-in procedures so you can perform complex editing tasks. See the *Guide to the DEC Text Processing*

Utility and the *DEC Text Processing Utility Reference Manual* for more information.

You also can customize your editing environment by customizing menus, and defining keys, tokens, placeholders, languages, templates, and aliases.

- Integrated programming environment

  LSE integrates the development environment by providing templates for several languages and by providing access to the other tools. This environment enables you to design, create, and edit code; view multiple source modules; compile programs; and review and correct compile-time errors in a single editing session. LSE integrates the development environment by providing templates for several languages and by providing access to the HP Source Code Analyzer (SCA), HP Code Management System (CMS), and OpenVMS Debugger.

  You can invoke LSE directly from your debugger to correct source code problems found during debugging sessions. On OpenVMS systems, you also can invoke LSE from PCA to correct performance problems. In addition, LSE and SCA are integrated to enable browsing of source code from within SCA.

- EVE/EDT keypads

  LSE provides a SET KEYPAD command that sets the key definitions to be similar to the EVE or EDT basic keypads.

- Online help

  LSE provides online help for information on unfamiliar language constructs and routines. Help is provided for all LSE commands and key definitions.

1.2 Getting Started

This section describes the LSE features to help you accomplish your tasks and includes the following topics:

- Task areas and menus
- Issuing commands
- Using help

1.2.1 Understanding LSE Concepts

Before you start using LSE, you need to understand the concepts of tokens and placeholders, which are language elements that have been predefined for each of the supported languages. You can expand these elements into templates for language constructs.

Tokens are reserved words or function names that you type into the editing buffer and expand to provide templates for corresponding language constructs. For example, you can type the keyword IF and then expand it into a complete skeleton IF statement, with consistent indentation and capitalization.

Placeholders are items surrounded by delimiters that are inserted into the editing buffer by LSE when you expand other placeholders or tokens. Placeholders are markers that indicate locations in the source code where you must provide additional program text, or choose from indicated options.

Placeholders are either required or optional. Required placeholders, indicated by braces ({}), represent places in the source code where you must provide program text. Optional placeholders, indicated by brackets ([]), represent places in the source code where you can either provide additional constructs or erase the placeholder. For example, a required placeholder might look like this:

```
{compilation_unit}
```

You can expand, erase, or type directly over placeholders. When you type over a placeholder, it is automatically removed and the text you type is inserted into the buffer. When you erase a placeholder, it is automatically deleted. However, if you erase a required placeholder, LSE displays a message saying that the placeholder is required, and asks if you want to continue the erase operation. When you press the EXPAND key (CTRL/E) while the cursor is on a placeholder, one of three events occurs:

- The placeholder is replaced automatically with a template consisting of language constructs. This type of placeholder is called a **nonterminal placeholder** because it inserts a template into the buffer when expanded.
- Text is displayed in a separate window to aid you in supplying a value. This type of placeholder is called a **terminal placeholder** because it does not insert a template into the buffer when expanded. Instead, you must supply the necessary text. You can press the spacebar to remove the window.
- A menu is displayed to provide you with options that can be selected and expanded into templates. This type of placeholder is called a **menu placeholder**.

In any of these three cases, you can type the desired text over the placeholder, and it is erased automatically. When expanding a menu placeholder, you can move through the options by using the up and down arrow keys. To select an option, press the EXPAND, RETURN, or ENTER key. To exit from the menu without selecting an option, press the spacebar.

Some placeholders are automatically duplicated when expanded. These placeholders are called **list placeholders**.

In addition, LSE provides **pseudocode placeholders**. Pseudocode placeholders contain natural-language text that expresses design information. Pseudocode placeholders, unlike regular placeholders, are not defined by LSE. LSE inserts pseudocode placeholder delimiters into the editing buffer when you press the ENTER PSEUDOCODE key (PF1-spacebar). You type the appropriate design information within the delimiters. You can move pseudocode placeholder text to program comments. See Chapter 3 for more details on using pseudocode.

You can construct a complete program by repeatedly expanding templates. You do not have to continuously expand templates until you reach a terminal placeholder. Rather, you might find it more appropriate to type in the desired value yourself at a higher level. See Chapter 3 for additional information on tokens and placeholders.

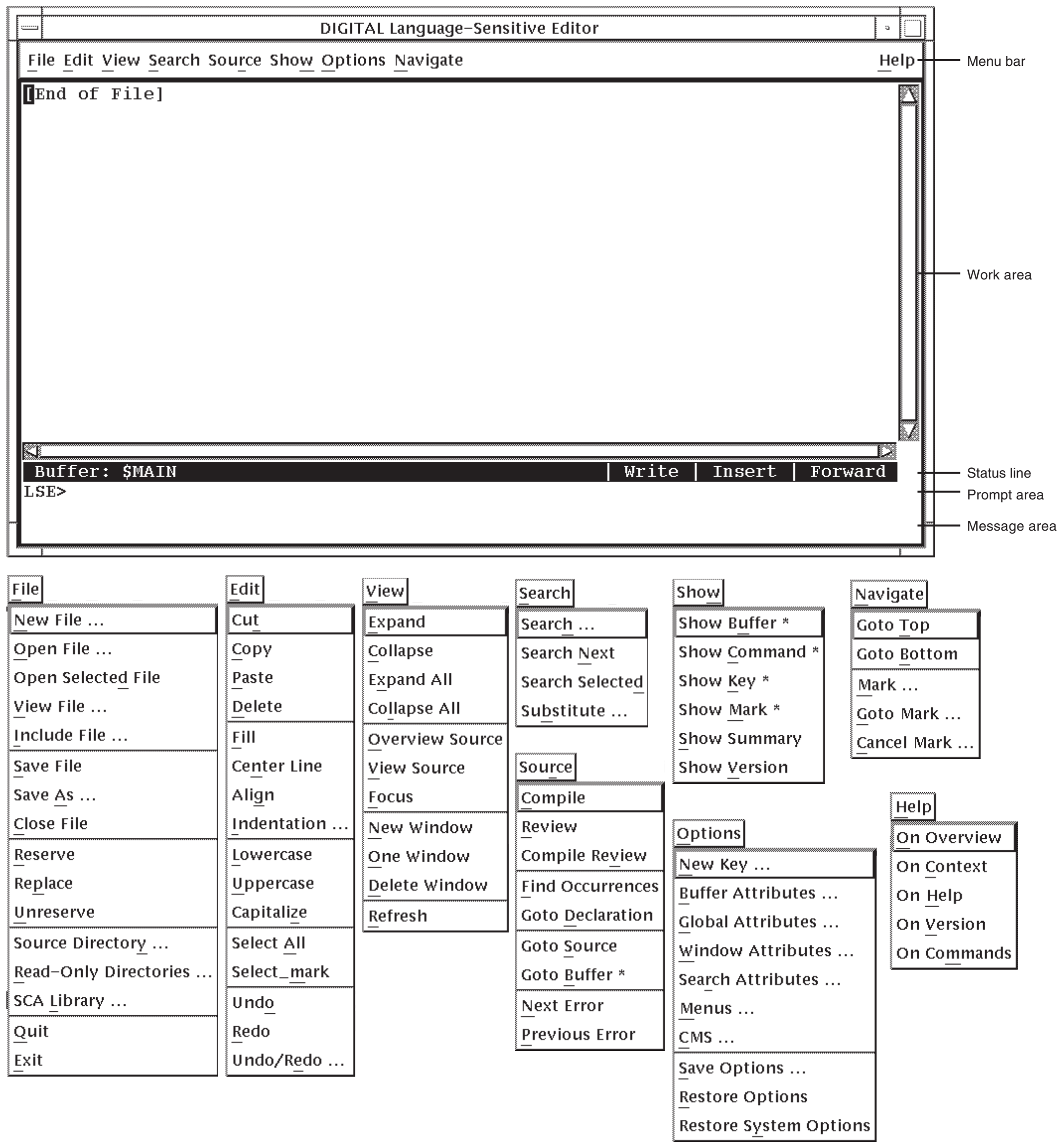
1.2.2 LSE Task Areas and Menus

The screen layout for LSE has the following task areas:

- Menu bar
- Work area
- Status line
- Prompt area
- Message area

Figure 1–1 shows a menu map of the LSE main session window menus with the task areas called out.

Figure 1–1 LSE Tasks Area and Menus



```
[End of File]

 Buffer: $MAIN                                    | Write | Insert | Forward
LSE>
```

DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Navigate Help
Menu bar
Work area
Status line
Prompt area
Message area
File: New File ... / Open File ... / Open Selected File / View File ... / Include File ... / Save File / Save As ... / Close File / Reserve / Replace / Unreserve / Source Directory ... / Read–Only Directories ... / SCA Library ... / Quit / Exit
Edit: Cut / Copy / Paste / Delete / Fill / Center Line / Align / Indentation ... / Lowercase / Uppercase / Capitalize / Select All / Select_mark / Undo / Redo / Undo/Redo ...
View: Expand / Collapse / Expand All / Collapse All / Overview Source / View Source / Focus / New Window / One Window / Delete Window / Refresh
Search: Search ... / Search Next / Search Selected / Substitute ...
Source: Compile / Review / Compile Review / Find Occurrences / Goto Declaration / Goto Source / Goto Buffer * / Next Error / Previous Error
Show: Show Buffer * / Show Command * / Show Key * / Show Mark * / Show Summary / Show Version
Options: New Key ... / Buffer Attributes ... / Global Attributes ... / Window Attributes ... / Search Attributes ... / Menus ... / CMS ... / Save Options ... / Restore Options / Restore System Options
Navigate: Goto Top / Goto Bottom / Mark ... / Goto Mark ... / Cancel Mark ...
Help: On Overview / On Context / On Help / On Version / On Commands

Table 1–1 describes the LSE task areas.

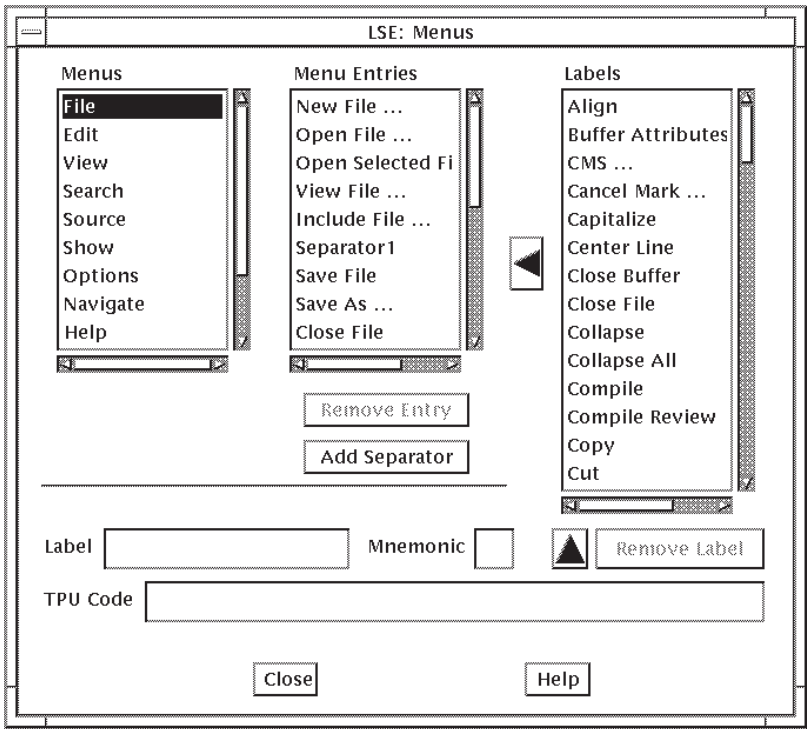
Table 1–1 LSE Window Task Areas

Task Area	Description
Menu bar	Provides all the HP DECwindows commands for LSE.
Work area	The editing buffers.
Status line	Provides information about the associated buffer, such as the name of the buffer, write or read-only status, insert or overstrike mode status, and forward or reverse direction status. Each window in the work area has its own status line.
Prompt area	Lets you enter commands.
Message area	Located at the bottom of the screen, the message area displays broadcast messages and messages issued by LSE.

You can customize your menus using the Menus dialog box, shown in Figure 1–2. You can rearrange existing menus or add new menu items to the LSE menus.

To access the Menus dialog box, choose the Menus ... item from the Options menu.

Figure 1–2 Customizing LSE Menus



1.2.3 Issuing Commands

LSE provides two ways to issue commands:

- Keypad mode
- Command-line mode

Using the keypad mode, you can also enter commands through keypad and control-key sequences. You can press keys to perform editing functions rather than typing commands on the command line. LSE binds commonly used commands to certain keys to simplify editing. To get help about the keypad, press PF1-F15 (PF1-Help), or enter the HELP KEYPAD command. To list the current key definitions, enter the SHOW KEY command.

Using the HP DECwindows interface, you can enter command functions from the LSE command line obtained by pressing F16 (Do) or PF1-KP7. Enter LSE commands at the command prompt that appears in the prompt area.

Some LSE commands are not bound to menu items or keys. Therefore, they must be entered in command line mode. There are two command-line prompts:

- LSE Command>
- LSE>

The LSE Command prompt processes one command. After that command is processed, LSE returns to keypad mode.

Alternatively, the LSE> prompt allows you to enter as many commands as you want. To get the LSE> prompt, press Ctrl/Z. The LSE> prompt appears near the bottom of the screen. To return to keypad mode, press Ctrl/Z again or enter the CONTINUE command.

For information about the LSE commands, choose On Commands from the Help menu. You can also type Help from the command line.

1.2.4 Using Command Languages

LSE has two command languages available: VMSLSE and Portable. The VMSLSE command language is the original LSE command language that has always been present in LSE, and has remained the most used. The Portable command language is a more recent command language devised for use in environments other than OpenVMS. The choice of command language is made in the HP DECset installation, but can be changed at any time.

See the *DIGITAL Language-Sensitive Editor Command-Line Interface and Callable Routines Reference Manual* for information on the following topics:

- Setting the default command language
- Invoking LSE command languages
- Using the SET PROMPT KEYPAD command
- Integrating LSE with SCA and CMS
- Integrating HP DECwindows LSE with HP DECwindows SCA

This guide presents examples of both VMSLSE and Portable commands.

1.2.5 Using Help

To obtain language-specific help on routines defined in packages, a particular keyword, or a placeholder, position the text cursor on the routine, keyword, or placeholder and press PF1, then PF2. Help is not available for all keywords and placeholders.

The HP DECwindows LSE user interface provides context-sensitive online help for window items, menu items, buttons, fields, and labels.

To get online help on any menu or submenu item, do the following:

1. Choose On Context from the Help menu. The pointer changes to a question mark.
2. Position the question mark on a window object and press MB1.

You can also get context-sensitive help as follows:

1. Set the location cursor by selecting the object.
2. Press the Help key.

If the input focus is in a dialog box and you press the Help key, you receive an overview of the entire dialog box, not just the selected item.

2
Using LSE

This chapter describes the following LSE operations:

- Invoking LSE
- Opening a file
- Manipulating text
- Using multiple windows
- Using buffers
- Defining key commands
- Defining key sequences
- Working with files
- Recovering from a failed editing session
- Storing custom modifications
- Automating LSE initialization

2.1 Invoking LSE

The format for invoking the LSE command line is as follows:

```
$ LSEDIT [/qualifiers] [file-spec]
```

LSEDIT invokes LSE; */qualifiers* specify command qualifiers; and *file-spec* specifies the file to be edited. It must be an OpenVMS file specification.

To invoke LSE using the HP DECwindows interface, enter the following command:

```
$ LSEDIT/INTERFACE=DECWINDOWS [file-specification]
```

If you do not specify a file name or file type in your file specification, LSE uses the file name or file type specified in your last LSEDIT command, if you issued the EXIT command to end that editing session. Otherwise, LSE creates a new buffer called $MAIN and prompts you for a file name when you exit from LSE, if you have added text to the $MAIN buffer.

You can also specify command options on the command line. These are described in the *DIGITAL Language-Sensitive Editor Command-Line Interface and Callable Routines Reference Manual*. However, the DECwindows interface provides customizing features that eliminate the need for command-line qualifiers or options after the features have been set up.

To exit from an editing session, choose Exit or Quit from the File menu. Choosing Exit saves the modifications you made during the editing session; choosing Quit does not save the modifications. DECwindows LSE can also be invoked from the DECset pull-down menu in the DECwindows Session Manager.

When you exit from an editing session, LSE stores information about the cursor position and the original input file so you can return to the same position in the file. However, this information is lost if you log out or invoke LSE from another process.

2.2 Opening a File

When you begin an editing session, you type the name of the file you want to edit in the LSEDIT command line. If you do not specify a file, LSE creates an empty buffer called $MAIN.

You can open a file using HP DECwindows in the following ways:

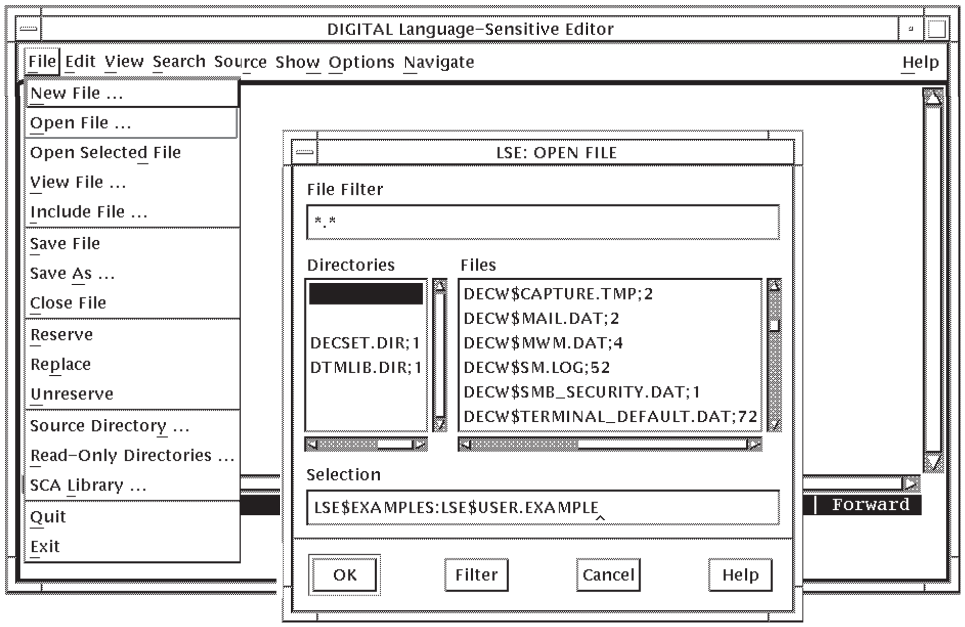
- Choose Open File . . . from the File menu to open a file that already exists. Specify the name of the file in the Open File dialog box.
- Choose New File . . . from the File menu to open a file that does not yet exist. Specify the name of the file in the New File dialog box.
- Select an existing file name in another window, then choose Open Selected File from the File menu.
- Type the name of a file after the LSEDIT command when you invoke LSE.
- Type the OPEN FILE or NEW FILE command.

To open the existing OpenVMS file LSE$EXAMPLES:LSE$USER.EXAMPLE, do the following:

1. Choose Open File . . . from the File menu. The OPEN FILE dialog box is displayed.
2. Specify LSE$EXAMPLES:LSE$USER.EXAMPLE in the Selection field.
3. Click on OK.

Figure 2–1 shows how to open this file.

Figure 2–1 Opening a File



File Filter: *.*

Directories:
DECSET.DIR;1
DTMLIB.DIR;1

Files:
DECW$CAPTURE.TMP;2
DECW$MAIL.DAT;2
DECW$MWM.DAT;4
DECW$SM.LOG;52
DECW$SMB_SECURITY.DAT;1
DECW$TERMINAL_DEFAULT.DAT;72

Selection: LSE$EXAMPLES:LSE$USER.EXAMPLE

2.3 Manipulating Text

This section describes some of the text editing functions of LSE, as follows:

- Selecting text for operations
- Searching for text using the SEARCH command
- Searching for text using DECwindows LSE
- Replacing text using the SUBSTITUTE command

- Replacing text using DECwindows LSE

You can position the cursor anywhere in the buffer by moving the pointer to the desired spot and pressing MB1. Each time you do this, you set the current cursor position to that location.

2.3.1 Selecting Text for Operations

You can select text by pressing MB1 in rapid succession as follows:

- Two clicks selects the current word.
- Three clicks selects the current line.
- Four clicks selects the current paragraph.
- Five clicks selects the current buffer.

To select text other than a word, line, paragraph, or buffer, press and hold MB1 and move the pointer to the end of the text you want to select.

When you select text, it becomes highlighted. You then choose an operation from the Edit menu. In most cases, the operation applies to all text that is selected.

To cancel a selection, make another selection or press MB1.

Table 2–1 describes the operations on the Edit menu that you can perform on selected text.

Table 2–1 Edit Menu Command Operations on Selected Text

Operation	Description
Cut	Deletes the selected text and puts it into the DECwindows clipboard (or the $paste buffer).[1] If you cut or copy another section of text before pasting the contents of the current clipboard in the editing buffer, the new section of text is placed in the clipboard and the previous text is lost.
Copy	Places the selected text into the clipboard (or the $paste buffer), without deleting it from the editing buffer.[1] If you copy another section of text before pasting the contents of the current clipboard in the editing buffer, the new section of text is placed in the clipboard and the previous text is lost.

[1]Controlled with either the DECwindows Global Attributes menu or the Portable command SET CLIPBOARD ON/OFF (usable in both command languages if prefixed by PLSE).

(continued on next page)

Table 2–1 (Cont.) Edit Menu Command Operations on Selected Text

Operation	Description
Paste	Places the contents of the clipboard (or the $paste buffer) into the editing buffer starting at the current text-cursor position.[1]
Delete	Deletes the selected text from the buffer. The text is not placed on the clipboard.
Fill	Fills the lines of a paragraph with text to the right margin. In programming language files, this operation applies only to comments and does not change the structure of the programming code. See Section 3.4 for more information.
Center Line	Centers the text on a line between the left and right margins. This operation applies only to the last line selected, even if you select more than one line of text.
Align	Aligns programming language comments under the first selected comment. This operation applies only to programming language-associated files. See Section 3.4 for more information.
Indentation ...	Moves blocks of text or code to the left or right in the buffer. If you click on Decrease successively in the Indentation dialog box, you can move all text to the left margin. Clicking on Increase moves the text to the right, but maintains the current indentation.
Lowercase	Changes any uppercase letters in the selected text area to lowercase letters.
Uppercase	Changes any lowercase letters in the selected text area to uppercase letters.
Capitalize	Changes all words in the selected area to an uppercase first letter, and lowercase letter on the letters that follow.
Select All	Places all the contents of the current buffer in the selected range. Any operations that LSE performs on a selected range then apply to all the contents of the buffer.

[1]Controlled with either the DECwindows Global Attributes menu or the Portable command SET CLIPBOARD ON/OFF (usable in both command languages if prefixed by PLSE).

Table 2–2 BOX Command Operations on Selected Text

Operation	Description
BOX COPY	Copies the currently selected box to the default location that can either be the HP DECwindows clipboard or the paste buffer.
BOX CUT	Cuts the selected box to the default location (the HP DECwindows clipboard or the paste buffer.)
BOX CUT/PAD	Indicates that the area of the cut is to be padded with spaces.
BOX PASTE	Copies the contents of the default location (the HP DECwindows clipboard or the paste buffer)to a box with its top left hand corner at the current position.
BOX PASTE/OVERSTRIKE	Copies the contents of the default location to a box with its top left hand corner at the current position. The contents are copied in the OVERSTRIKE mode.
BOX DRAW	Draws a box in the OVERSTRIKE mode. The box is drawn using the plus sign (+) for the corners, the vertical bar | for the sides and a hyphen (-) for the top and bottom.
BOX LOWERCASE	Changes the case of the text in the selected box to lowercase.
BOX_UPPERCASE	Changes the case of the text in the selected box to uppercase.

The following restrictions apply:

- A box operation is not allowed if there is overview information in the records that make up the box.
- If a box operation is performed for which the total size of the records containing the box before or after the operation is greater than 65535 characters, then the Undo information is reset.
- Auto Erase is switched off during a box paste operation.
- Tabs in the records containing the box are converted to spaces and the records may be extended with spaces.

2.3.2 Searching for Text Using the SEARCH Command

The SEARCH command searches through the current buffer for a specified text string in either a forward or reverse direction, depending on the current direction setting. If an occurrence of the search string is found, the cursor is positioned on the first character of the text string and the text string is highlighted. If the string is not found, the cursor is not moved and an error message appears in the message buffer.

LSE keeps a search text string until you end the editing session so you can search again for the same string by entering the SEARCH or SUBSTITUTE command and pressing Return. You can search for a text string used during the editing session by entering the SEARCH or SUBSTITUTE command, using the up arrow key to find a previously used text string, and pressing Return.

LSE supports pattern search operations using the SEARCH command with the /PATTERN qualifier. For example, the asterisk (*) character can be used to match any number of characters within one line. The percent sign (%) character can be used to match any one character.

If you want to search for an asterisk (*) or percent sign (%) as part of your pattern, you must include a backslash (\ ) before using the asterisk (*) or percent sign (%) in the pattern search string, enclosed in quotation marks.

For example, if you want to search for 20%, 22%, and so on, in a buffer, enter the following command:

```
LSE> SEARCH/PATTERN "2%\%"
```

2.3.3 Searching for Text Using DECwindows LSE

The Search menu lets you search for text in the following ways:

- Choose Search . . . from the Search menu and specify a text string in the Search dialog box. Clicking on Apply finds the next (or previous) text string that matches starting from the current cursor position; clicking on OK does the same thing, but dismisses the Search dialog box.
- Choose Search Next to find the next occurrence (or previous occurrence, depending on the search direction) of the most recent text string specified in the Search dialog box.
- Select a text string and choose Search Selected to find the next occurrence (or previous occurrence, depending on the search direction) of the selected text string.

- Choose Substitute . . . to search for a text string and replace it with another text string. Section 2.3.5 describes how to replace text using DECwindows LSE.

Choose Search Attributes . . . from the Options menu to set such search attributes as spanning spaces in strings, case sensitivity, diacritic sensitivity, and automatic searches of text in the opposite direction of the search. You can also choose OpenVMS system pattern searches.

2.3.4 Replacing Text Using the SUBSTITUTE Command

The SUBSTITUTE command replaces occurrences of one text string with another text string. When you enter the SUBSTITUTE command, LSE prompts you for the search string and the value of the replacement string.

For example, you can use the SUBSTITUTE command to replace all occurrences of APPEND with PREFIX, as follows:

```
LSE> SUBSTITUTE APPEND PREFIX
```

LSE provides case-sensitive substitution with the SUBSTITUTE command and /EXACT qualifier, which lets you specify that the case of the replacement string should be altered to match the case of the string located by a search operation.

For example, you can search for the APPEND string name and replace it with PREFIX by entering the following command:

```
LSE> SUBSTITUTE/EXACT APPEND PREFIX
```

LSE alters the case of `prefix` to match the case of `append` found in your file.

LSE highlights each occurrence and prompts you for an action, as follows:

- Yes (or pressing Return) instructs LSE to replace the occurrence.
- No instructs LSE not to replace the occurrence, but to search for the next occurrence.
- Quit ends the command without replacing the occurrence and stops the SUBSTITUTE operation.
- Last instructs LSE to replace the occurrence and then ends the command.
- ALL replaces the occurrence and all remaining occurrences without further prompting.

If you use the SUBSTITUTE command with the ALL keyword, LSE replaces all occurrences it finds without prompting you for an action.

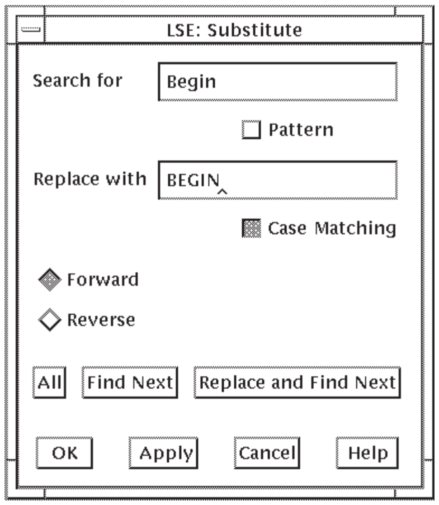
You can use the SUBSTITUTE command with the /PATTERN qualifier for pattern replacement operations using the same pattern expressions as those with the SEARCH command. The following example shows how to use the SUBSTITUTE/PATTERN command:

```
LSE> SUBSTITUTE/PATTERN "NAME_%_LENGTH" "NAME_B_LENGTH"
```

2.3.5 Replacing Text Using DECwindows LSE

Choose Substitute . . . from the Search menu. Figure 2–2 shows how to replace the text string Begin with BEGIN with case sensitivity in the Substitute dialog box.

Figure 2–2 Substitute Dialog Box



To replace all subsequent occurrences of a string from the text cursor position, click on the Forward button and click on All.

To replace the next occurrence of a string, click on Apply or OK; clicking on OK dismisses the Substitute dialog box.

If you want to review each occurrence before the replacement is made, click on Find Next to find the next occurrence of the specified text string. Click on Apply or OK to confirm replacement, or click on Find Next to skip that occurrence of the text string.

2.4 Using Multiple Windows

You can display multiple windows during an LSE session. A **window** is a section of your work region that displays the contents of a buffer. When you create a new window, LSE divides the work area evenly in the session window. You can expand the work area by using the mouse pointer to expand the session window.

To create a new window, choose New Window from the View menu.

To move the text cursor to another window, click on the other window where you want to edit.

The following commands manipulate windows:

DELETE WINDOW
GOTO BUFFER
NEW WINDOW
NEXT WINDOW
OPEN FILE
PREVIOUS WINDOW
SET BALANCE WINDOWS ON or OFF
SET HEIGHT
SET MAXIMUM WINDOWS
SET MINIMUM WINDOW LENGTH
SET NUMBER OF WINDOWS
SET WIDTH
TWO WINDOWS

2.5 Using Buffers

LSE uses **buffers**, which hold information. There are two kinds of buffers: **system buffers** and **user buffers**. System buffers provide information about your editing session and LSE status. User buffers let you type input into a work area and can be stored in a file when you exit and save your input.

2.5.1 System Buffers

LSE uses system buffers for special purposes. Unlike user buffers, system buffers do not correspond to files. You can edit a system buffer like any other buffer, but you should avoid changing its contents. By convention, system buffer names start with a dollar sign ($). System buffers are displayed using the SHOW SYSTEM BUFFER command.

The most frequently used system buffers are $DEFAULTS, $HELP, $MESSAGES, $REVIEW, and $SHOW. The following list briefly describes each of these buffers:

- $DEFAULT—The $DEFAULT buffer contains the default attributes for any buffer you create in the current editing session. You can modify certain default attributes by entering LSE commands (for example, SET BUFFER LEFT MARGIN) from within the $DEFAULT buffer. Subsequent buffers that you create will assume these defaults.
- $HELP—The $HELP buffer displays help information when you enter a HELP command.
- $MESSAGES—The $MESSAGES buffer displays information about operations performed while in an editing session (for example, notification that a file has been written).
- $REVIEW—When you enter the REVIEW command, the $REVIEW buffer displays sets of diagnostic messages that result from a compilation.
- $SHOW—The $SHOW buffer displays output from any SHOW command (for example, SHOW BUFFER).

2.5.2 User Buffers

LSE sets the user buffer attributes and properties for each newly created buffer. To override these settings by using an initialization file, do the following:

1. Create an initialization file in which to put your commands.
2. When you want these buffer attributes, enter the following command:

   ```
   $ LSEDIT/INITIALIZATION=init_filename filename
   ```

You can modify the attributes of the $DEFAULT buffer. When a new buffer is created, it adopts the attributes in the $DEFAULT buffer.

The following commands set buffer attributes:

```
SET BUFFER AUTO_ERASE
SET BUFFER CLOSE
SET BUFFER DIRECTION
SET BUFFER INDENTATION
SET BUFFER LANGUAGE
SET BUFFER LEFT MARGIN
SET BUFFER MODIFIABLE
SET BUFFER OUTPUT
SET BUFFER RIGHT MARGIN
```

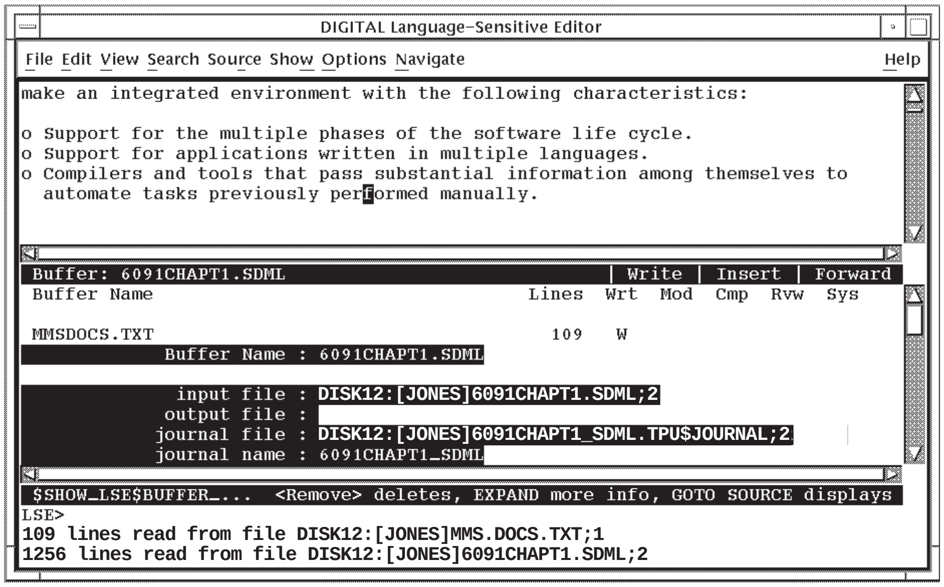
SET BUFFER TAB INCREMENT
SET BUFFER TEXT
SET BUFFER WRAP

There are several cases where these attributes are overridden, as follows:

- If you can determine a language from the file type, the attributes for that language are in effect.
- If a newly created buffer is associated with a defined language, the left margin, right margin, tab increment, and wrap attributes defined for that language are used.

Figure 2–3 shows how to display buffer information by choosing Show Buffer * from the Show menu. All buffers from the current LSE session are listed in the split screen.

Figure 2–3 LSE Buffers


DIGITAL Language-Sensitive Editor
File Edit View Search Source Show Options Navigate
Help

```
make an integrated environment with the following characteristics:

o Support for the multiple phases of the software life cycle.
o Support for applications written in multiple languages.
o Compilers and tools that pass substantial information among themselves to
  automate tasks previously performed manually.

 Buffer: 6091CHAPT1.SDML                               | Write | Insert | Forward
 Buffer Name                                   Lines  Wrt  Mod  Cmp  Rvw  Sys

 MMSDOCS.TXT                                     109   W
             Buffer Name : 6091CHAPT1.SDML

              input file : DISK12:[JONES]6091CHAPT1.SDML;2
             output file :
            journal file : DISK12:[JONES]6091CHAPT1_SDML.TPU$JOURNAL;2
            journal name : 6091CHAPT1_SDML
 $SHOW_LSE$BUFFER_...  <Remove> deletes, EXPAND more info, GOTO SOURCE displays
LSE>
109 lines read from file DISK12:[JONES]MMS.DOCS.TXT;1
1256 lines read from file DISK12:[JONES]6091CHAPT1.SDML;2
```

As the example shows, you can expand the information about a buffer by double clicking on it in the $SHOW_LSE$BUFFER buffer. Note that the 6091CHAPT1.SDML source is displayed in the upper window, whereas the 6091CHAPT1.SDML buffer information is expanded in the lower window. In addition, the MMSDOCS.TXT buffer shows only two values in its collapsed view. Double clicking again collapses the information.

You can also examine a buffer's contents by selecting it and choosing Goto Source from the Source menu. In addition, you can revert to the collapsed view by again choosing Show Buffer * from the Show menu.

Table 2–3 lists the numerous LSE buffer attributes.

Table 2–3 Buffer Attributes

Attribute	Description
buffer name	A buffer has a name that is displayed in the status line. Buffers are usually named by the name and type of their associated input file. The OPEN FILE, NEW FILE, and NEW BUFFER commands can create buffers that you can edit.
input file	Specifies the name of the file in the buffer. The OPEN FILE command creates a buffer and reads a file into it. The INCLUDE FILE command reads an existing file into the current buffer.
output file	By default, the output file has the same name as the input file. You can write the buffer to another file name by entering the SET BUFFER OUTPUT command, which will change the property of this attribute.
journal file	Specifies the DECTPU-defined name of the buffer in which all keystrokes and buffer actions are stored during your editing session.
journal name	Same name as the buffer.
safe for journaling	A buffer is safe for journaling if it is empty, has never been modified, or has not been modified since the last time it was written to a file. You can set this attribute using built-in procedures.
journaling	Sets whether all keystrokes and buffer actions are stored in the journal file during your editing session.
language	This attribute determines which language is used for the language-sensitive features. The file type of the input file associated with your current buffer determines the language LSE uses. You can move between different languages in different buffers, and LSE will provide the interfaces to the appropriate compilers. The SET BUFFER LANGUAGE command associates a language with a buffer.
key map list	This attribute is informational only and shows the name of the key map list. It is modifiable only through DECTPU programming.
modified	Specifies whether any changes have been made to the buffer.

(continued on next page)

Table 2–3 (Cont.) Buffer Attributes

Attribute	Description
modifiable	Unmodifiable buffers protect the contents of a given buffer. You cannot change an unmodifiable buffer. If you want to modify an unmodifiable buffer, you must enter the SET BUFFER MODIFIABLE ON command. There are some relationships between the close attributes (read-only and save) and the modifiable buffer attributes. Given these attributes, a buffer can be in one of four possible states. The following list describes these states and explains how to create these states for a buffer: • Modifiable On and Close Save The SET BUFFER CLOSE SAVE command sets buffers to this state. It is also the default for the file specified in the LSEDIT command line. The buffer can be modified and is written when you exit from the editing session, if it has been modified. • Modifiable On and Close Read-Only This is the default for the NEW BUFFER command that you use to create a scratch buffer. This temporary work buffer can be modified, but it is not written when you exit from the editing session. • Modifiable Off and Close Read-Only The SET BUFFER CLOSE READ_ONLY command sets buffers in this state. The buffer cannot be modified. If you enter a SET BUFFER MODIFIABLE ON command on this buffer and modify the contents, LSE does not save the contents when you exit from the editing session, unless you also enter the SET BUFFER CLOSE SAVE command for the buffer. • Modifiable Off and Close Save You can set a buffer to this state when you have completed a set of changes to a buffer in the modifiable and writable state and then entered a SET BUFFER MODIFIABLE OFF command for the buffer. This protects the buffer from accidental change for the remainder of the editing session. LSE saves the file when you exit from the editing session, if it has been changed during the session.

(continued on next page)

Table 2–3 (Cont.) Buffer Attributes

Attribute	Description
unmodifiable record	Shows whether the buffer contains one or more unmodifiable records.
close	Buffers have either the read-only or save property. The close attribute indicates that the contents of the buffer are not written to a file when you exit from the editing session. The close attribute set to save indicates that the buffer is written to a file when you exit from the editing session. Enter the SET BUFFER CLOSE command to change this attribute. Usually, a file is associated with a buffer by the OPEN FILE command, which creates a buffer and fills it with the contents of a file. When the buffer is written, it is written to the file. If no file is associated with a buffer that has the save attribute, LSE prompts for a file specification when you exit from the editing session. A buffer is written only if its contents have been modified.
erase unmodifiable	Specifies whether you can erase unmodifiable records.
text	LSE has two text entry modes: insert and overstrike. In insert mode, text is inserted into the buffer at the cursor position. Text to the right of the cursor moves to the right. In overstrike mode, text typed at the cursor replaces text that is currently under the cursor. When you start an editing session, the buffer is automatically placed in insert mode. To change the text entry mode, you can use the SET BUFFER TEXT INSERT or SET BUFFER TEXT OVERSTRIKE command.
direction	LSE maintains a current direction for each buffer. The current direction is displayed in the status line. This direction is used for search operations and most GOTO and ERASE commands. When you start an editing session, the buffer direction is set to forward by default. To set the current direction to forward, use the SET BUFFER DIRECTION FORWARD command. To set the current direction to reverse, use the SET BUFFER DIRECTION REVERSE command. Alternatively, you can use the CHANGE DIRECTION command to change the current direction.

(continued on next page)

Table 2–3 (Cont.) Buffer Attributes

Attribute	Description
indentation level	LSE maintains two settings to control the action of the Tab key: current indentation level and tab increment. When you are at the left margin, the Tab key indents to the current indentation level. If you are not at the left margin, the Tab key takes you to the next tab column based on the tab increment setting. The SET BUFFER INDENTATION command sets the current indentation level; the SET BUFFER TAB INCREMENT command sets the size of the tab increment.
auto erase	Using the SET BUFFER AUTO ERASE command, if the auto erase attribute is set to ON, LSE erases the placeholder the cursor is on as soon as you insert a character over the placeholder.
overviews	The SET BUFFER OVERVIEW ON command enables the COLLAPSE, FOCUS, and VIEW SOURCE commands, and the use of the EXPAND command on an overview line.
wrap	Using the SET BUFFER WRAP command, if the wrap attribute is set to ON, LSE automatically inserts a carriage return and indents in the left margin when the text reaches the right margin.
record size	Shows the maximum length for records (lines) in the buffer.
record count	Shows the number of records currently in the buffer.
record number	Shows the record number on which the cursor is positioned.
LSE left margin	The SET BUFFER LEFT MARGIN command sets the left margin for the indicated buffer. By default, the LSE left margin is set at column 1. The LSE left margin affects features specific to LSE.
TPU left margin	By default, the DECTPU left margin is set at column 1. The DECTPU left margin affects features specific to DECTPU.
right margin	The SET BUFFER RIGHT MARGIN command sets the right margin for the indicated buffer to the column number you specify. By default, the right margin is set at column 80. The right margin controls where LSE wraps words when you type text into a buffer. The right margin also controls how the FILL command reformats text. The right margin is the same for LSE and DECTPU.
offset	Specifies the number of characters between the left margin and the cursor position. The left margin is counted as position 0. A tab is counted as one character, regardless of width. Window shifts have no effect on the value of the offset.

(continued on next page)

Table 2–3 (Cont.) Buffer Attributes

Attribute	Description
offset column	Specifies the column in which DECTPU displays the character at the cursor position. This value reflects the location of the left margin and the width of tabs preceding the cursor position. Window shifts have no effect on the value of the offset column; DECTPU treats a window that is in the current buffer as not shifted.
tab increment	LSE maintains two settings to control the action of the Tab key: current indentation level and tab increment. When you are at the left margin, the Tab key indents to the current indentation level. If you are not at the left margin, the Tab key takes you to the next tab column based on the tab increment setting. The SET BUFFER INDENTATION command sets the current indentation level; the SET BUFFER TAB INCREMENT command sets the size of the tab increment.
tab stops	Shows the column numbers where tab stops are set.
window count	Shows the number of windows in which the current buffer is displayed.

2.6 Defining Key Commands

With LSE, you can associate any command or keystroke sequence with a key or control-key combination on the keyboard. LSE provides default command definitions for some keys, such as the key-command association for the keypad (press PF2 to see these). You can redefine the default command definitions.

To see the current definition of all the keys, choose Show Key * from the Show menu.

To define a key-command association, choose New Key ... from the Options menu and specify the required information on the New Key dialog box.

You can also define a key interactively by using the NEW KEY command. You must include the name of the key you want defined and the command string you want bound to that key. (For key names, see the NEW KEY command description in the online Help on LSE, in the “On Commands” area.)

For example, if you want to define a key to exit from LSE, enter the following command:

```
LSE> NEW KEY F9 "EXIT"
```

Now you can press F9 to exit from this editing session.

The following example shows how you can define the NEW KEY command to execute whenever you press Ctrl/A:

```
Command: NEW KEY
_Key: Ctrl/A
LSE Command: NEW KEY
```

After this definition, pressing Ctrl/A immediately displays the _Key: prompt. If you do not know the name of a key, such as Ctrl/A, you can automatically display the name of a key by pressing PF1-Ctrl/V and then the key you want to define.

Note

You can use PF1-Ctrl/V only if you first enter the SET PROMPT KEYPAD command.

In addition, you can bind several commands to a key. To do this, you must use the DO command and place quotation marks around each command. The following example shows how to bind the ENTER LINE and TAB commands to the Return key (specified by CTRL/M_KEY):

```
DEF KEY CTRL/M_KEY "DO ""ENTER LINE"",""ENTER TAB"""
```

When you issue this command, position the cursor at the end of the line above where you want to insert text, and press the Return key. A blank line is inserted and the cursor is placed at the proper depth for your code.

2.7 Defining Key Sequences

As an alternative to binding commands to a key, you can bind a sequence of keystrokes to a key. This is called a **learn sequence**. To begin recording a learn sequence, enter the following command:

```
LSE> NEW LEARN KEY
```

At the prompt, press the key that you want to define, F17 for example. LSE echoes the key name.

Next, press the Return key to start recording keystrokes. Every key that you press now will be recorded. This includes all keys that enter text into the editing buffer. All commands typed at the LSE Command> prompt and all responses to prompts such as “_Search for:” are also recorded.

To end the sequence, press the key that you are defining, in this case F17. Pressing F17 again replays the learn sequence.

Pointer actions are not recorded.

2.8 Working with Files

This section describes the following file-manipulation operations:

- Locating, displaying, and writing source files
- Locating files in multiple directories
- Setting directory defaults
- Rolling back the last LSE function performed
- Getting files through the HP Code Management System
- Editing features with the HP Source Code Analyzer System

2.8.1 Locating, Displaying, and Writing Source Files

LSE provides the following commands for locating and displaying source files within your editing sessions:

Command	Description
GOTO DECLARATION	Displays the source file corresponding to the specified or indicated symbol declaration.
INCLUDE FILE	Inserts a copy of a file at the current cursor position. The cursor position does not change.
GOTO SOURCE	Displays the source file corresponding to the current diagnostic or query item.
OPEN FILE	Locates a file and reads it into a buffer.
NEW FILE	Creates a new file in a buffer.

LSE provides the following commands for writing the contents of buffers into files:

Command	Description
COMPILE	First saves the current buffer (if it is not read-only), if it has been modified, and saves any other modified buffers associated with the same language. It then compiles the file associated with the current buffer.

Command	Description
EXIT	Ends an editing session and saves buffers that have been modified, provided they are not marked as read-only buffers. Buffers that are read-only are not written out by an EXIT command. (If you do not want to save your modifications, use the QUIT command to end your editing session.)
SAVE FILE	Saves your current buffer or a specified buffer in a file using the current file name. Your editing session continues until you enter an EXIT or QUIT command.
SAVE AS	Saves your current buffer or a specified buffer in a file you specify. Your editing session continues until you enter an EXIT or QUIT command.

2.8.2 Locating Files in Multiple Directories

You can specify a list of directories for LSE to use when locating files by using the command SET DIRECTORY SOURCE. For example, type the following command:

```
LSE> SET DIRECTORY SOURCE DIR 1 DIR 2 DIR 3 DIR 4
```

LSE searches the directories in the order specified on the SET DIRECTORY SOURCE command line.

When you specify a file on the OPEN FILE command line, LSE resolves it by using the search list established with the SET DIRECTORY SOURCE command.

Note

The directory search list cannot contain commas. Substitute the appropriate operating-system-specific directory specifications for DIR 1, DIR 2, DIR 3, and DIR 4 in the previous command example.

LSE searches through directories DIR 1 through DIR 4, in order, until the source file is found.

You can include CMS$LIB in the SET DIRECTORY SOURCE directory list so LSE will fetch files directly from your CMS library into buffers. The CMS SET LIBRARY command must be entered first.

The following commands use the source list to locate files:

GOTO SOURCE
GOTO DECLARATION

INCLUDE FILE
OPEN FILE

The GOTO SOURCE command displays a source file that is specified in the diagnostics file, or SCA data that corresponds to your current review or query operation. If LSE cannot find the file within an existing buffer, it tries to find the exact file specified by the current diagnostic or query operation. If that file cannot be found, LSE uses the SET DIRECTORY SOURCE list to locate the file.

The GOTO DECLARATION command displays the source file for the declaration of the specified or indicated symbol.

The INCLUDE FILE command also uses the SET DIRECTORY SOURCE list to resolve file specifications. The INCLUDE FILE command inserts the contents of a file into the buffer at the current cursor position. The cursor is positioned on the first character of the inserted text, rather than remaining on the original character, or at the end of the buffer.

2.8.3 Setting Directory Defaults

With the SET DIRECTORY commands, you can set the default CLOSE status of files in a specified directory. Initially, all directories are set to save.

The SET DIRECTORY CLOSE READ_ONLY *directory* command enables you to specify directories containing files that you do not want to change. If you specify the READ_ONLY keyword, LSE brings the files contained in those directories into unmodifiable, read-only buffers.

The SET DIRECTORY CLOSE SAVE *directory* command allows you to modify and write to files.

Using Default Settings

If you are working on a software project, you might have the following directories set up:

- *cur_dir*—Your current directory.
- *my_dir*—Contains the files you are working on, such as MODULE1.PAS and MODULE3.PAS.
- *proj_dir*—Contains the files that comprise your project, for example, MODULE1.PAS, MODULE2.PAS, MODULE3.PAS, MODULE4.PAS, and MODULE5.PAS. These files are shared by your project team and must not be modified.

For example, enter the following commands to set your default settings:

```
SET DIRECTORY SOURCE cur_dir my_dir proj_dir
SET DIRECTORY CLOSE proj_dir READ_ONLY
```

Then, enter an OPEN FILE or GOTO SOURCE command. LSE searches through *cur_dir*, *my_dir*, and *proj_dir* to locate files. Those files located in your current directory and *my_dir* can be modified. However, those files located in *proj_dir* cannot be modified by default because of the READ_ONLY keyword on the SET DIRECTORY command. Thus, you can modify files MODULE1.PAS and MODULE3.PAS, and refer to the other project files without the possibility of modifying the wrong files.

The following example gets the file from *proj_dir* and puts the file into an unmodifiable, read-only buffer:

```
OPEN FILE MODULE2.PAS
```

The following example gets the file from *my_dir* and puts the file into a modifiable, writable buffer:

```
OPEN FILE MODULE1.PAS
```

Overriding Default Settings

The GOTO FILE command has the following qualifiers:

- /WRITE
- /READ_ONLY
- /MODIFY

You can use these qualifiers to override any settings established by the SET DIRECTORY command. (See the GOTO FILE command in the *HP DECset for OpenVMS Language-Sensitive Editor / Source Code Analyzer Reference Manual* for more details.) You can also use the /WRITE and /READ_ONLY qualifiers on the GOTO SOURCE command to override any settings established by the SET DIRECTORY command.

If you want to modify MODULE2.PAS in the [PROJECT_DIRECTORY], enter the following command:

```
LSE> GOTO FILE/WRITE MODULE2.PAS
```

LSE gets MODULE2.PAS out of the PROJECT_DIRECTORY and puts the file into a modifiable/writable buffer.

```
LSE> SET WRITE
```

The GOTO FILE/WRITE command sets a buffer to both modifiable and writable. Thus, you could issue the following command if you find that you need to make changes to MODULE4.PAS that you want to keep:

```
LSE> GOTO FILE/WRITE MODULE4.PAS
```

Because the file is located in a /READ_ONLY directory, LSE issues an information message to that effect.

On the other hand, you might find that you do not want to modify MODULE1.PAS, even though it is not located in a READ_ONLY directory.

If you want to look at MODULE1.PAS in MY_DIRECTORY, but you do not want to modify the file, enter the following command:

```
LSE> GOTO FILE/READ_ONLY MODULE1.PAS
```

This command overrides the current settings established by the previous SET DIRECTORY command, and it brings MODULE1.PAS into a read-only/unmodifiable buffer. If you want to modify the file, issue the following command:

```
LSE> SET MODIFY
```

2.8.4 Rolling Back the Last LSE Function Performed

The LSE operations or commands that might need undoing fall into the following categories:

- Internal:
  - Those that affect settings
  - Those that affect buffers
- External:
  - Those that affect external resources (disk files)

LSE does not have complete control over external resources. For this reason, operations or commands that influence external resources cannot be undone. Examples of such commands are as follows:

- LSE WRITE
- LSE CMS RESERVE

The LSE operations or commands that can be rolled back are limited to those that modify the user buffers.

The following UNDO and REDO commands are implemented in both character-cell and DECwindows interfaces to facilitate the rolling back of LSE operations or commands. Note that these operations are limited to the current user buffer only.

UNDO Command

Multiple UNDO commands reverse previous buffer modifications in the current user buffer, one at a time. Multiple UNDO commands are by default restricted to reversing the last 100 buffer modifications per user buffer. When this limit is reached in the current buffer, the UNDO command is disabled in the DECwindows interface. The UNDO command is reenabled only after another command is executed (including a REDO command) in the current user buffer.

You can use the SET MAX_UNDO command to set the number of maximum allowable UNDO commands for the current buffer (or the specified buffer) to a number other than the default value of 100, as follows:

```
LSE> SET MAX_UNDO[/buffer=buffer_name] number
```

For more information on the UNDO command, see the *HP DECset for OpenVMS Language-Sensitive Editor / Source Code Analyzer Reference Manual*.

REDO Command

Multiple REDO commands reverse the effects of previous UNDO commands in the current user buffer. In the case where all previous UNDO commands have been redone for the current user buffer, the REDO command is disabled in the DECwindows interface. The REDO command is reenabled only after another UNDO command is executed in the current user buffer.

For more information on the REDO command, see the *HP DECset for OpenVMS Language-Sensitive Editor / Source Code Analyzer Reference Manual*.

Note

Enable the global UNDO (and REDO) feature by setting SET MODE UNDO=ON, and disable them with the command SET MODE UNDO=OFF. For details, see the HP Language-Sensitive Editor/Source Code Analyzer for OpenVMS Reference Manual.

2.8.5 Getting Files Through HP Code Management System

With LSE, you can access files stored in HP Code Management System (CMS) directly. You can also issue all CMS commands from within LSE. The LSE commands related to CMS are as follows:

- CMS [cms-command]
- SET CMS
- GOTO FILE
- GOTO SOURCE
- INCLUDE
- READ
- RESERVE
- UNRESERVE
- REPLACE

CMS [cms-command] Command

With the CMS [cms-command] command, you can execute any CMS command from within LSE. This command operates on your current CMS library.

SET CMS Command

The SET CMS command sets the defaults for CMS qualifier values for CMS operations performed by the following commands: GOTO FILE, GOTO SOURCE, INCLUDE, READ, REPLACE, RESERVE, and UNRESERVE.

GOTO FILE, GOTO SOURCE, INCLUDE, and READ Commands

When you issue the GOTO FILE, GOTO SOURCE, INCLUDE, or READ command, if the directory for a file you are looking for is the same as your current CMS library, LSE fetches the element from CMS and puts it into a read-only/unmodifiable buffer. LSE prompts you for confirmation when performing a fetch operation. This fetch operation does not create a file.

For example, enter the following command and response to locate and fetch a CMS element:

```
LSE> GOTO FILE CMS$LIB:FILE.TYP
FILE.TYP found in CMS library DISK:[PROJECT_CMS_LIBRARY]
Do you want to fetch it [Yes]: Y
```

LSE fetches the element FILE.TYP and reads it into a buffer of the same name. LSE does not create a file in your current directory when doing a fetch operation.

RESERVE Command

With the RESERVE command, you can reserve an element in your current CMS library. This element is put into an editing buffer. Enter the following command:

```
LSE> RESERVE element-name
```

LSE reserves the element *element-name* in your current CMS library and reads it into a buffer. If you omit the *element-name* parameter, the RESERVE command reserves the element of the same name and type as the input file associated with your current buffer.

UNRESERVE Command

With the UNRESERVE command, you can unreserve the CMS element of the same name and type as the file associated with your current buffer in your current CMS library. Enter the following command:

```
LSE> UNRESERVE
```

LSE unreserves the element in your current buffer that you reserved in the last example.

REPLACE Command

The REPLACE command replaces the CMS element of the same name and type as the file associated with your current buffer in your current CMS library. Enter the following command:

```
LSE> REPLACE
```

LSE replaces the element in your current buffer into your current CMS library.

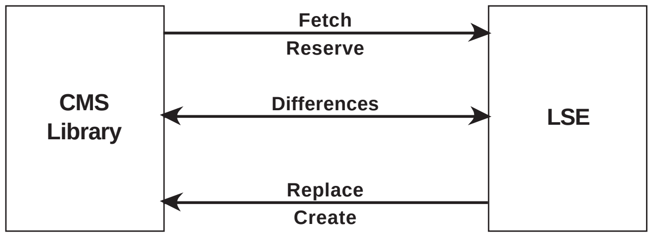
2.8.6 DECwindows LSE/CMS Integration

LSE is integrated with CMS to ease the management of source code between the two DECset components, as follows:

- From within LSE, enter commands or select menu choices to manipulate CMS elements, LSE buffers, or disk files.
- From within CMS, select menu choices to manipulate CMS elements, LSE buffers, or disk files.

As shown in Figure 2–4, you can fetch and reserve an element from a CMS library, edit the file or perform differences, or replace or create a file to the library.

Figure 2–4 LSE/CMS Integration



2.8.6.1 CMS Functions from LSE

From LSE, you can perform all CMS operations on the CMS library of your choice. To see which library is set, issue the following command:

```
LSE> CMS SHOW LIBRARY
```

The following example shows a typical system response that might appear in the LSE buffer:

```
Your CMS library list consists of:
   DISK11:[EXCERPTS.CMS.VMS]
```

To reset to another CMS library, issue a command similar to the following:

```
LSE> CMS SET LIBRARY DISK11:[SUMMARIES.CMS.VMS]
```

The following list further describes the major CMS functions available from LSE integrated functions, and presents examples:

- Reserve (and unreserve) an element in the CMS library into an LSE buffer. Use the LSE File menu or the LSE command line. The following example shows how to reserve generation 12 of a CMS element at the LSE command line:

  ```
  LSE> CMS RESERVE COPY.PAS/GENERATION=12
  ```

- Replace an element into the CMS library from an LSE buffer. Use the LSE File menu or the LSE command line. The following example shows how to replace a CMS element at the LSE command line:

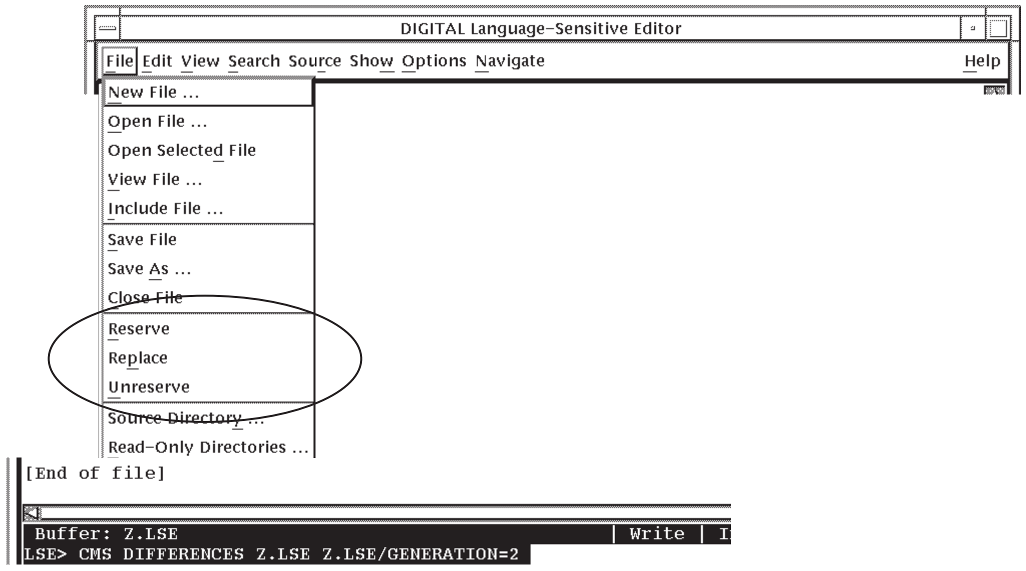
  ```
  LSE> CMS REPLACE COPY.PAS "Nov 1998 update"
  ```

- Perform CMS differences between any combination of CMS element and disk file, putting the results into a disk file. Use the LSE command line only. The following example shows how to perform differences between generation 15 of a CMS element and version 2 of the related disk file at the LSE command line:

```
LSE> CMS DIFFERENCES COPY.PAS/GENERATION=15 COPY.PAS;2
```

Figure 2–5 shows the CMS commands available from the LSE File menu. The sample LSE command line shows a CMS DIFFERENCES operation between the current LSE buffer and generation 2 of the related CMS element.

Figure 2–5 Performing CMS Functions from LSE



```
[End of file]
 Buffer: Z.LSE                                    | Write | I
LSE> CMS DIFFERENCES Z.LSE Z.LSE/GENERATION=2
```

All other CMS operations are available via the LSE command-line interface.

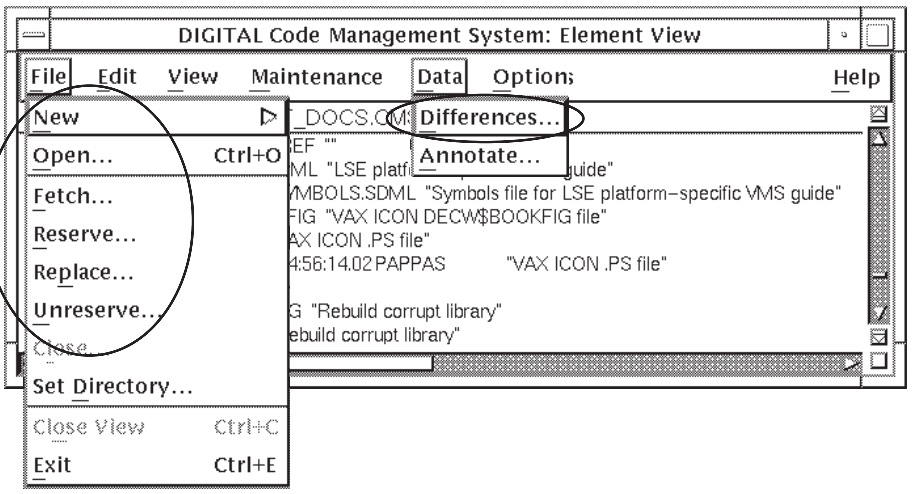
2.8.6.2 LSE Functions from CMS

From the CMS File pull-down menu, you can do the following:

- Create an element in the CMS library from the current LSE buffer.
- Fetch a generation of an element from the CMS library into an LSE buffer.
- Reserve an element in the CMS library into an LSE buffer.
- Replace an element into the CMS library from an LSE buffer.
- Perform CMS differences between two generations of elements.
- Perform CMS differences between the current LSE buffer and its corresponding CMS element or a file.

Figure 2–6 shows the CMS commands available for integration with LSE. Only one pull-down menu can be activated at a time.

Figure 2–6 Performing LSE Functions from CMS



```
_DOCS.CM
EF ""
ML "LSE platf          guide"
MBOLS.SDML  "Symbols file for LSE platform-specific VMS guide"
FIG  "VAX ICON DECW$BOOKFIG file"
X ICON .PS file"
4:56:14.02 PAPPAS          "VAX ICON .PS file"

G  "Rebuild corrupt library"
ebuild corrupt library"
```

2.8.6.3 Creating an Element in the CMS Library

To create a CMS element, use one of the following methods:

- **Create a CMS element using LSE.**
  To create a CMS element from the LSE command line, issue a command similar to the following:

  ```
  LSE> CMS CREATE ELEMENT COPY.PAS "Dec 1998 update"
  ```

  If the specified file is in the current LSE buffer and is different from the latest version on disk, a new version is saved and that version is used for the CMS element.

- **Create a CMS element using CMS.**
  You can create an element in the CMS library from the contents of the current LSE buffer via the CMS pull-down menu. If the specified file name is currently in an LSE buffer, the LSE buffer is written to disk and that version of the CMS element is created.

To create an element using the CMS menu, do the following:

1. From the CMS File menu, choose New, Element to access the New Element dialog box.
2. Specify an LSE edit buffer in the Element text field, or accept the current LSE edit buffer (if LSE is running). The buffer name must be a valid CMS element name, as defined in the *Guide to HP Code Management System for OpenVMS Systems*. If the text in the current LSE buffer has changed from an older version on disk, a new version is written to disk at the same time that the new element is created in the CMS library.

   You can also import a file by clicking the Input File toggle button and typing a file name in the associated text field. The Input File and Element choices are mutually exclusive. That is, you can specify either an input file or an LSE edit buffer, but not both.

2.8.6.4 Fetching a Generation of an Element From the CMS Library

To fetch a CMS element, use one of the following methods:

- **Fetch a CMS element using LSE.**
  To fetch a CMS element in LSE, either select Fetch from the File menu, or issue a command similar to the following at the command line:

  ```
  LSE> CMS FETCH COPY.PAS "Modify Dec 1998 update"
  ```

  To fetch an earlier generation, specify the /GENERATION=*n* qualifier. If DECwindows LSE is running, the specified CMS element is fetched and displayed in an LSE buffer with the same name. For a different file name, use the /OUTPUT=*file-spec* qualifier.
- **Fetch a CMS element using CMS.**
  You can fetch a generation of an element from the CMS library into an LSE buffer via the CMS File menu, or by double clicking on the element name. If DECwindows LSE is running, the specified CMS element is fetched and displayed in an LSE buffer with the same name.

2.8.6.5 Reserving an Element in the CMS Library

To reserve (or unreserve) a CMS element, use one of the following methods:

- **Reserve a CMS element using LSE.**
  To reserve a CMS element in LSE, either select Reserve from the File menu, or issue a command similar to the following at the command line:

  ```
  LSE> CMS RESERVE COPY.PAS "Modify Dec 1998 update"
  ```

To reserve an earlier generation, specify the /GENERATION=*n* qualifier. If DECwindows LSE is running, the specified CMS element is reserved and displayed in an LSE buffer with the same name. For a different file name, use the /OUTPUT=*file-spec* qualifier.

- **Reserve a CMS element using CMS.**
  You can reserve an element in the CMS library into an LSE buffer via the CMS File menu. If DECwindows LSE is running, the specified CMS element is reserved and displayed in an LSE buffer with the same name.

2.8.6.6 Replacing an Element into the CMS Library

To replace a CMS element, use one of the following methods:

- **Replace a CMS element using LSE.**
  To replace a CMS element in LSE, either select Replace from the File menu, or issue a command similar to the following at the command line:

  ```
  LSE> CMS REPLACE COPY.PAS "Oct 1998 update, mod. Dec 1998"
  ```

  To avoid creating a new generation if the input file has no changes from the reserved generation, use the /IF_CHANGED qualifier.

- **Replace a CMS element using CMS.**
  You can replace an element into the CMS library from an LSE buffer via the CMS File menu. If the specified file name is currently in an LSE buffer, the LSE buffer is written to disk and that version of the CMS element is replaced. Do the following:

  1. From the CMS File menu, choose Replace to access the Replace dialog box.
  2. Verify the selected CMS elements in the Selected list box, change element names or generations, or click on Cancel to return to the CMS window. If you did not select an element in the CMS window, the cursor appears in the Element text field for your input. However, if you did select an element, the Element text field is unavailable.

  To avoid creating a new generation if the input file has no changes from the reserved generation, activate the Create New Generation Only if Changed toggle button in the Replace Options dialog box.

2.8.6.7 Performing CMS Differences Operations

To perform CMS differences operations, use one of the following methods:

- **Perform CMS differences from LSE.**
  Enter the command and specify which items to use to perform the operation. Enter only the file name for an LSE buffer, add version numbers for disk files, or add the /GENERATION=*n* qualifier for CMS elements, as shown in the following examples:
  - The first example performs differences between generation 2 of a CMS element and version 8 of a disk file.

    ```
    LSE> CMS DIFFERENCES COPY.PAS/GENERATION=2 COPY.PAS;8
    ```
  - The second example performs differences between generation 2 of a CMS element and generation 1 of the same CMS element.

    ```
    LSE> CMS DIFFERENCES COPY.PAS/GENERATION=2 COPY.PAS/GENERATION=1
    ```
- **Perform CMS differences from CMS.**
  Access the Differences dialog box from the Data pull-down menu in DECwindows CMS. Use the Primary Input region to identify the first item to be compared, and the Secondary Input region for the second item. Select the two items with the following buttons and fields:
  - **Selected**—If you selected a CMS element, that element appears in the Selected field of the Primary Input region. You can either accept that element, change its generation number, or select another element or file. If you did not select a CMS element, the Selected area is inactive.

    By default, CMS uses the highest mainline generation (1+), unless you selected a specific generation. To compare any other generation, supply the exact generation number in the form ELEMENT\n.
  - **Generation**—To compare a CMS element, click the Generation toggle button and enter the generation value in the text field.
  - **Element/File**—Click on the Element/File label and specify either an OpenVMS file specification or a CMS element. The OpenVMS file can be specified without a version number, but a CMS element *must* be specified with a generation number. For a CMS element, click the Generation button to specify that the file is an element and not an OpenVMS file. If no CMS element was selected, the text field remains blank in both the Primary Input and Secondary Input areas.

2.8.7 Editing Features with HP Source Code Analyzer System

LSE is tightly integrated with SCA. All SCA commands are available within LSE. LSE/SCA enables you to design, create, compile, correct, and inspect source code within a single editing session. For example, to modify the characteristics of a variable, you can use SCA to locate all of the uses of the variable across the system and make the changes without leaving LSE.

You can perform SCA queries within LSE by combined use of the LSE command prompt (command-line interface) and the basic LSE navigational commands (HP DECwindows interface). Alternatively, you can use the LSE command-line interface exclusively. For example, you can use the LSE command prompt to set the SCA library and query the library for occurrences of symbols. You can then use basic LSE navigational commands (point and click operations via MB1) to examine the corresponding source code for a particular occurrence. You can then use the query buffer pop-up menu to move between items and source. In addition, you can use the LSE command prompt in place of any basic LSE navigational command.

The LSE and SCA windows must be displayed to the same X server but do not need to be running on the same node. For example, LSE can be running locally while SCA runs remotely on a server system.

For more information on LSE or SCA features or integration, see *Using HP DECset for OpenVMS Systems*.

2.9 Recovering From a Failed Editing Session

LSE provides mechanisms to recover edits that exist as changes in LSE buffers when a system or editor failure occurs. LSE can journal your edits in two ways: keystroke journaling and buffer-change journaling. Keystroke journaling records the keys that you press over the course of an editing session. Buffer-change journaling records the changes made to a buffer over the course of an editing session.

Keystroke journal files have a file type of .TJL and buffer-change journal files have a file type of .TPU$JOURNAL. Both types of journaling periodically save information to their respective journal files. When an editing session is terminated abnormally, a journal file might not contain a record of the last few operations that were performed. When you recover using a journal file, your cursor remains at the position corresponding to the location where the last journaled edit was made.

Using a Keystroke Journal File

Keystroke journal files contain the exact sequence of keys that you pressed over the course of the editing session. When you recover using a keystroke journal file, you must be sure to restore your environment to the state that it was in when the journaled editing session was started. The following list describes the types of things that must be restored to your environment before you can successfully complete a recovery using a keystroke journal file:

- All files created during the editing session must be either deleted or placed in a directory that will not be referenced during the recovery operation. This will prevent the wrong version of the file from being accessed during the keystroke-journal recovery operation.

  You can determine what files were created during an editing session by examining the creation date of the .TJL file. For example, to determine the creation date of a keystroke journal file, enter the following command:

  ```
  $ DIRECTORY/DATE=CREATE MEMO.TJL
  ```

  This produces output similar to the following:

  ```
  Directory DUA0:[SMITH]

  MEMO.TJL;1         15-JUN-1998 08:42:20.69

  Total of 1 file.
  ```

  To move files from specific directories that were created since that date to a directory that is not referenced during the recovery operation, enter the following commands:

  ```
  $ CREATE/DIRECTORY DUA0:[SMITH.TEMP]
  $ COPY/SINCE=15-JUN-1998:08:42:20 -
  _$ DUA0:[SMITH...],DUA0:[PROJECT...] DUA0:[SMITH.TEMP]
  $ DELETE/SINCE=15-JUN-1998:08:42:20 -
  _$ DUA0:[SMITH...],DUA0:[PROJECT...]
  ```

- You must set the terminal characteristics of the terminal to match the terminal characteristics that were in effect at the time the journaled editing session was started.

- You must invoke LSE with the same command line that you used to start the journaled editing session. Additionally, you must specify the /RECOVER qualifier on the command line.

You might have to take other actions, depending on the events that might have occurred over the course of your editing session.

If you reserve or replace elements in a CMS library during the editing session, you might not be able to perform a recovery using a keystroke journal file because changes in the CMS library cannot be undone. In such cases, use a buffer-change journal file.

Using a Buffer-Change Journal File

A buffer-change journal file contains a record of the changes that you made to a buffer over the course of an editing session. There is one buffer-change journal file for each editing buffer.

To recover changes with a buffer-change journal file, do the following:

1. Invoke LSE.
2. Enter the RECOVER BUFFER command and specify the name of the file that you want to recover.
3. Examine the information displayed by LSE to verify that the journal file corresponds to the source file you want.
4. Enter Y at the prompt if you want to recover the buffer.

After you instruct LSE to start the recovery using a buffer-change journal file, LSE reads the source file that corresponds to the buffer-change journal file and begins to apply the journaled changes to the file.

When you recover changes using a buffer-change journal file, you do not have to worry about the files that were created during the editing session, or the command line that you used to invoke LSE. Using a buffer-change journal file is much quicker than using a keystroke journal file because LSE recovers only files that it is directed to recover.

2.10 Storing Custom Modifications

The previous sections explained how to make modifications interactively. Such modifications remain in effect only for the current editing session. If you want to keep your modifications, put them in a file and access that file whenever you want to use the modifications. Alternatively, you can create an initialization file or a command file that allows you to automatically access the modifications at LSE startup.

To store your modifications in a file, create a text file with a file type of .LSE. Similarly, if you want to keep your HP DECTPU statements, create a text file with a .TPU file type.

With LSE, you can use these files interactively when you need them. The following examples show how you execute your commands. The EXECUTE BUFFER command directs LSE to execute the commands found in the specified buffer. In both examples, the current buffer is used.

Press Ctrl/Z to get the command prompt, and enter the following commands:

For LSE commands:

```
$ NEW FILE filename.LSE
$ EXECUTE BUFFER LSE
```

For HP DECTPU statements:

```
$ NEW FILE filename.TPU
$ EXECUTE BUFFER TPU
```

The commands contained in the text file that you execute with the EXECUTE BUFFER commands affect the current editing session.

2.10.1 Using Initialization and Command Files on OpenVMS Systems

You might want your modifications automatically executed when you invoke LSE. You can do this by specifying either an initialization or command file when you invoke LSE. An initialization file contains LSE commands, whereas a command file contains HP DECTPU statements.

To use your LSE modifications, enter the following command:

```
$ LSEDIT/INITIALIZATION=device:[directory]filename.LSE ...
```

To use your HP DECTPU file, enter the following command:

```
$ LSEDIT/COMMAND=device:[directory]filename.TPU ...
```

You can use both the /INITIALIZATION and /COMMAND qualifiers on the LSEDIT command line to use both your files.

You can also use your initialization and command files without specifying the files each time you invoke LSE. To execute the initialization and command files each time you issue the LSEDIT command, add the following commands to your LOGIN.COM file.

```
$ DEFINE LSE$INITIALIZATION device:[directory]filename.LSE
$ DEFINE LSE$COMMAND device:[directory]filename.TPU
```

Example 2–1 shows the types of modifications you can put into an initialization file.

Example 2–1 Sample Initialization File

```
!Sample initialization file
!
! Command to spawn a subprocess, run MAIL, and clear the message
! buffer when the editing session is resumed.
!
DEFINE COMMAND MAIL "DO ""SPAWN MAIL"",""CLEAR_MESSAGE"""
!
! Command to clear the message window by writing three blank lines
!
DEFINE COMMAND CLEAR_MESSAGE-
  "DO/TPU ""MESSAGE("""""""")"",""MESSAGE("""""""")"",""MESSAGE("""""""")"""
!
! Command to bind the MAIL command to a key.
!
NEW KEY F20 "MAIL"
!
! DEFINE ALIAS command for abbreviating the name of an include file in Pascal.
!
DEFINE ALIAS DEFS/LANGUAGE=PASCAL "USER1:[PROJECT]COMMON_DEFINITIONS.PAS"
!
! DEFINE TOKEN command to redefine the DO token in C, so that it always contains a
! compound statement
!
DEFINE TOKEN DO -
  /LANGUAGE=C -
  /DESCRIPTION="executes a statement as long as a particular condition is satisfied" -
  /TOPIC="Language_topics Statements do"

  "do"
  "{"
  "    {@statement@}..."
  "}"
  "while    ({@expression@});"

END DEFINE
!
! HP DECTPU statement to change the scrolling of LSE windows so that the
! screen only scrolls when the cursor is at the bottom or top of the screen.
!
SET SCROLL_MARGINS 00
```

Example 2–2 shows the types of HP DECTPU procedures you can put into a command file. Procedure definitions must precede statements in a command file.

Example 2–2 Sample Command File

```
! Sample command file
!
! HP DECTPU procedure to replace tabs with spaces. In this example, the tabs
! are set at 8.
PROCEDURE ELIMINATE_TABS

    local       target,
                n,
                saved_mode;
    position(beginning_of(current_buffer));
    loop
        target := search(ascii(9), FORWARD);
        exitif (target = 0);
        position(beginning_of(target));
        erase_character(1);
        n := current_offset;
        n := n - (8 * (n / 8));
        saved_mode := get_info (current_buffer, "mode");
        set (insert, current_buffer);
        copy_text(substr("       ", 1, 8 - n));
        set (saved_mode, current_buffer);
    endloop;
ENDPROCEDURE

! LSE command to invoke the ELIMINATE_TABS procedure.
LSE$DO_COMMAND("DEFINE COMMAND NOTABS ""CALL ELIMINATE_TABS""");
! Command to change the text of the "Working..." message to "BUSY..."
SET (TIMER, ON, "BUSY...");

! HP DECTPU procedure to change the scrolling of LSE windows so that the screen
! only scrolls when the cursor is at the bottom or top of the screen.
LSE$DO_COMMAND "SET SCROLL_MARGINS 00"
```

2.11 Automating LSE Initialization

LSE provides two mechanisms to automate LSE initialization:

- An environment file that contains all language-specific definitions
- A section file that contains all key definitions and HP DECTPU procedures

Because both environment and section files are binary files, LSE does not have to execute them each time LSE is invoked. This saves time at LSE startup.

In addition, environment files and section files have a mechanism for a system manager to provide users with a tailored environment while still allowing you to do some customization of your own with the use of command files and initialization files.

If your initialization file or command file is large, you might want to put your changes into an environment file or a section file. If you do, save your source files for future LSE updates because LSE requires you to rebuild your environment and section files when you install a new version of LSE.

2.11.1 Creating Environment and Section Files

To create an environment or section file, establish the language definitions, placeholder definitions, command and key definitions, and mode settings that you want to save. Then, enter the SAVE ENVIRONMENT or SAVE SECTION command while in an editing session.

The SAVE ENVIRONMENT command saves the following:

languages	routines
packages	parameters
placeholders	adjustments
tokens	keywords
aliases	tags

The SAVE SECTION command saves the following:

key definitions	pending delete
cursor free/bound	HP DECTPU procedures and variables
learn sequences	bell
user-defined commands	bell broadcast
search attributes	keypad
balance windows	tabs visible
minimum window size	height
width	

You must include the name of the environment or section file, including device and directory names, when you enter the SAVE ENVIRONMENT or SAVE SECTION command.

The following examples show you how to create environment and section files. Press Ctrl/Z to get the command prompt and enter the following commands:

To create environment files:

```
$ NEW FILE LANGUAGE_DEFINITION.LSE
$ EXECUTE BUFFER LSE
$ SAVE ENVIRONMENT filename
```

To create section files:

```
$ NEW FILE TPU_PROGRAM.TPU
$ EXECUTE BUFFER TPU
$ SAVE SECTION filename
```

2.11.2 Using Environment and Section Files

To use your definitions, you must inform LSE about them by issuing the /ENVIRONMENT or /SECTION qualifiers on the LSE command line.

To use your environment file, enter the following command:

```
$ LSEDIT/ENVIRONMENT=device:[directory]filename.ENV
```

You can include a list of file specifications with the /ENVIRONMENT qualifier.

To use your section file, enter the following command:

```
$ LSEDIT/SECTION=device:[directory]filename.TPU$SECTION
```

You can automatically access all language-specific definitions from one editing session to another without specifying the /ENVIRONMENT qualifier each time you invoke LSE. To do this, add the following command to your LOGIN.COM file:

```
$ DEFINE LSE$ENVIRONMENT device:[directory]filename.ENV
```

You can automatically access all your key definitions and HP DECTPU procedures from one editing session to another without specifying the /SECTION qualifier each time you invoke LSE. To do this, add the following command to your LOGIN.COM file:

```
$ DEFINE LSE$SECTION device:[directory]filename.TPU$SECTION
```

2.11.3 Using Multiple Files

With LSE, you can specify any combination of the /INITIALIZATION, /COMMAND, /ENVIRONMENT, and /SECTION qualifiers on the command line. LSE processes these files in the following order:

1. /SECTION
2. /COMMAND
3. /ENVIRONMENT
4. /INITIALIZATION

If you specify more than one environment file with the /ENVIRONMENT qualifier, the definitions in the first file on the list take precedence over definitions in subsequent environment files. All environment file definitions take precedence over any administrator-supplied/SYSTEM_ENVIRONMENT file definitions.

Table 2–4 lists where your modifications to LSE are stored.

Table 2–4 Where LSE Stores Modifications

Text Files		Binary Files	
Initialization	**Command**	**Environment**	**Section**
LSE commands	HP DECTPU statements	Language-specific definitions	Key definitions
			HP DECTPU procedures
			HP DECTPU symbol names
			DEFINE COMMAND definitions
			Learn sequences
			LSE mode settings

3 Programming with LSE

This chapter describes the following topics:

- Compiling and reviewing a program
- Using tokens and placeholders
- Creating pseudocode
- Formatting comments
- Outlining programming code

3.1 Compiling and Reviewing a Program

While writing your program, you can use the COMPILE and REVIEW commands to compile your code and review compilation errors without leaving the editing session. Supported OpenVMS compilers generate a file of compile-time diagnostic information that LSE can use to review compilation errors. The diagnostic information is generated with the /DIAGNOSTICS qualifier.

The COMPILE command issues a DCL command in a subprocess to invoke the appropriate compiler. LSE checks to see if the language supports diagnostics capabilities. If so, LSE appends the /DIAGNOSTICS qualifier to the COMPILE command.

For example, if you issue the COMPILE command while in the buffer TEST.ADA, the resulting DCL command is as follows:

```
$ ADA DEV:[DIRECTORY]TEST.ADA/DIAGNOSTICS=DEV:[DIRECTORY]TEST.DIA
```

LSE supports all of the compiler's command qualifiers, as well as user-supplied command procedures. You can specify DCL qualifiers, such as /LIBRARY, when invoking the compiler from LSE. In addition, you can specify the /DESIGN qualifier to process designs. For example, to invoke the compiler with the /DESIGN qualifier, enter the following command:

```
LSE>  COMPILE /DESIGN
```

The REVIEW command displays any diagnostic messages that resulted from a compilation. LSE displays the compilation errors in one window, with the corresponding source code displayed in a second window.

Multiwindow capability allows you to review your errors while examining the associated source code. This eliminates tedious steps in the error correction process and helps ensure that all the errors are fixed before looping back through the compilation process.

LSE provides several commands to help you review errors and examine your source code. Table 3–1 lists these commands and their functions. Default key bindings are also shown where available.

Table 3–1 Commands for Reviewing Compilation Errors

Command	Function
COMPILE/REVIEW	Compiles the contents of a buffer and then displays a set of diagnostic messages that results from the compilation.
COMPILE	Compiles the contents of a buffer.
REVIEW	Selects and displays a set of diagnostic messages that results from a compilation.
END REVIEW	Ends an LSE REVIEW session.
GOTO SOURCE	Displays the source corresponding to the current diagnostic item. Use Ctrl/G on EDT, EVE VT100, and EVE VT200 keypads.
NEXT STEP	Moves the cursor forward to the next error. Use Ctrl/F on EDT, EVE VT100, and EVE VT200 keypads.
PREVIOUS STEP	Moves the cursor back to the previous error. Use Ctrl/B on EDT, EVE VT100, and EVE VT200 keypads.

The following example demonstrates how to compile and review errors within your editing session. This example and the sample language, called EXAMPLE, do not have compiler support. Therefore, when experimenting with the COMPILE command, construct your own example using one of the languages that has compiler support.

If you are editing a program and you want to compile it, do the following:

1. Press Ctrl/Z for the LSE> prompt and enter the COMPILE command.

   While the compilation is running, you can continue to use LSE to edit. When the compilation ends, a message is displayed in the message buffer.

2. Enter the REVIEW command at the LSE> prompt.

   The REVIEW command instructs LSE to review the compilation errors generated by the COMPILE command.

3. Press Ctrl/Z to move from the LSE> prompt to the buffer and begin reviewing the errors.

   You can use the NEXT STEP key (Ctrl/F) and the PREVIOUS STEP key (Ctrl/B) to step from error to error in the $REVIEW buffer. Any cursor movement keys can be used to examine the contents of the $REVIEW buffer. If you want to correct the source code associated with a particular error, press the GOTO SOURCE key (Ctrl/G) while the cursor is positioned on that error.

   In addition to the NEXT STEP and PREVIOUS STEP commands, you can use any cursor-movement keys to move around in the $REVIEW buffer. You can enter the GOTO SOURCE command to go to the associated source code. Thus, you can examine other lines of code associated with an error.

4. Press Ctrl/G (the GOTO SOURCE key) to correct the source code associated with the first error.

   Corrections accompany some errors. You can accept or reject the correction.

5. Enter Y after the following prompt to accept the correction:

   ```
   Keep the indicated correction [Y OR N]?
   ```

6. Press Ctrl/F (the NEXT STEP key) to move to the next error.

7. Press Ctrl/G (the GOTO SOURCE key) to display source lines with this error in the $REVIEW buffer.

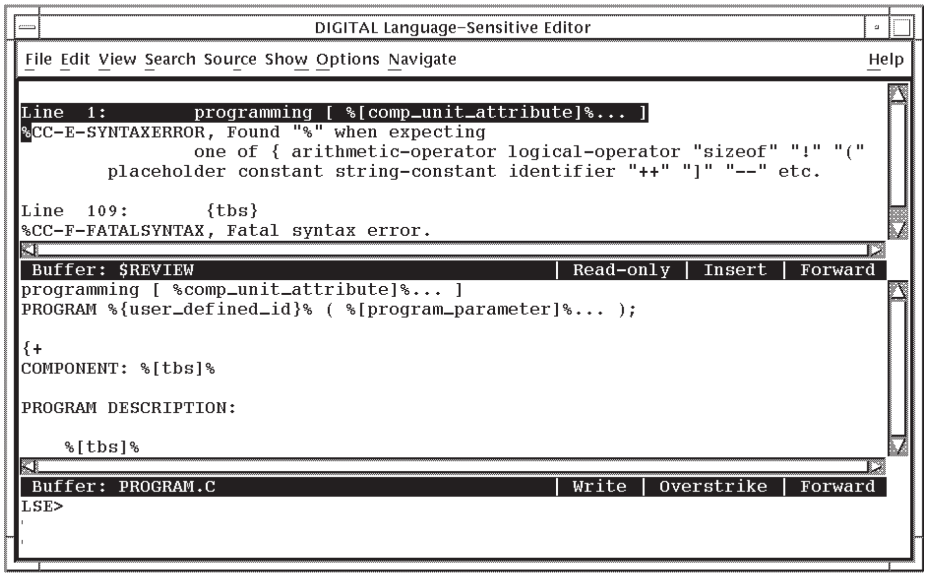
8. Press PF1-up arrow (the PREVIOUS WINDOW key) to go to the $REVIEW buffer.

9. Press the up arrow key to move the cursor to lines within the $REVIEW buffer.

10. Press Ctrl/G (the GOTO SOURCE key) to correct the source code associated with the next error. When you finish correcting your program, you can return to a single window containing the source code.
11. Press Ctrl/Z to get the LSE> prompt and enter the END REVIEW command.

To compile your program using the HP DECwindows interface, choose Compile Buffer from the Source menu. If errors are indicated in the message window, choose Review Buffer from the Source menu. Figure 3–1 shows the review and text buffers after compiling the text buffer and reviewing the error status.

Figure 3–1 Compiling and Reviewing a Program


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Navigate Help

```
Line  1:        programming [ %[comp_unit_attribute]%... ]
%CC-E-SYNTAXERROR, Found "%" when expecting
                one of { arithmetic-operator logical-operator "sizeof" "!" "("
        placeholder constant string-constant identifier "++" "]" "--" etc.

Line  109:      {tbs}
%CC-F-FATALSYNTAX, Fatal syntax error.
 Buffer: $REVIEW                                | Read-only | Insert | Forward
programming [ %comp_unit_attribute]%... ]
PROGRAM %{user_defined_id}% ( %[program_parameter]%... );

{+
COMPONENT: %[tbs]%

PROGRAM DESCRIPTION:

    %[tbs]%
 Buffer: PROGRAM.C                              | Write | Overstrike | Forward
LSE>
```

If you double click on an error message in the review buffer, LSE automatically places the text cursor on the corresponding line in the text buffer.

3.1.1 Invoking LSE from OpenVMS Debugger and from HP Performance and Coverage Analyzer

While in the OpenVMS Debugger (debugger) or the HP Performance and Coverage Analyzer (PCA), you can invoke LSE to edit your code.

The command syntax to invoke LSE from the debugger or PCA is as follows:

```
DBG> EDIT [/EXIT] [[module-name\] line-number]
```

LSE positions the cursor at the line in the file that corresponds to the specified module and line in the debugger or PCA. The default file and line are taken from the current source display.

The rules for specifying the module-name and line-number qualifiers are as follows:

- If both the *module-name* and *line-number* are specified, their values determine the module and line at which the file is positioned in LSE.
- If only the *line-number* is specified, the module is assumed to be the module of the current source display.
- If neither the *module-number* nor *line-number* is specified, the module is assumed to be the module of the current source display, and the line number is assumed to be the central line in the window of that display.

When you invoke LSE from the debugger or PCA, a subprocess is spawned for the editing session. Control automatically returns to the debugger or PCA after the editing is completed.

You use the edit command to tell LSE exactly what file you are editing. The EDIT command also checks the file in the debugger or the PCA source display to see if it is the most recent version. LSE always edits the most recent version of the file. If the displayed version is not the most recent version, LSE issues an error message.

When debugging and editing code, it is faster to use the EDIT/EXIT command at the DBG> prompt instead of returning to the debugger and typing the EXIT command. This is useful when only minor editing must be done before recompiling.

3.2 Using Tokens and Placeholders

LSE provides predefined programming language elements, called tokens and placeholders, for each of the supported programming languages. You can expand these elements into templates for language constructs. For information about the predefined programming-language elements, see Chapter 4.

Tokens are typically keywords in programming languages. When expanded, tokens provide additional language constructs. Tokens are typed directly into the buffer. Generally, tokens are used in situations when you want to add additional language constructs where there are no placeholders. For example, typing IF and pressing the EXPAND key causes a template for an IF construct to appear on your screen. Tokens are also used to bypass long menus in situations where expanding a placeholder, such as {statement}, would result in a lengthy menu.

You can use tokens to insert text when editing an existing file. Because most languages have tokens for built-in functions and keywords, you enter the name for a function or keyword and press the EXPAND key. In addition, most languages provide a token named *statement* or *expression* that expands into a menu of all possible statements or expressions.

The following example demonstrates how to use tokens to edit an existing program. In this case, the buffer TEST.EXAMPLE contains the following code:

```
PROCEDURE test ()

    IF A = B
    THEN
        A = C + 1
    ENDIF
ENDPROCEDURE test
```

If you want to add more statements to this program before the IF construct, do the following:

1. Move the cursor to the beginning of the IF statement line.
2. Press the OPEN LINE key (PF1-KP0).
3. Press the Tab key.

   Note that the cursor is placed at the same level of indentation as the IF statement line.
4. Type *statement* and press the EXPAND key.

In the menu of statements now displayed, use the arrow keys to scroll through the menu. To select a menu item, press the EXPAND, Enter, or Return key. Press the spacebar to exit from a menu without selecting an item.

Pseudocode placeholders are placeholders that contain natural language text that expresses design information.

Pseudocode is easy to write and provides a way to sketch your design ideas. You can convert pseudocode into comments, thus providing a way to preserve design information. You can use a compiler to process pseudocode to detect syntax errors and generate SCA data.

For example, the following code fragment contains pseudocode delimited by the special brackets « and » defined by the DEFINE LANGUAGE command:

```
procedure my_proc (file_name : in out my_type) is
begin
    «read the file»;
    if «the file is empty» then
        «clean up»;
        «stop»;
    else
        «process the file»;
    end if;
    put_message;
end my_proc;
```

Pseudocode placeholders can appear only in well-defined places, such as in place of statements, expressions, and declarations; otherwise, the compiler will issue an error message. The following example shows pseudocode placeholders used in place of statements:

```
procedure do_something is
begin
    «Open the file.»
    «Read and process records.»
    «Write the file.»
    «Close the file.»
end do_something;
```

The following examples show pseudocode used in place of expressions:

```
if «it is a leap year»

while «there is snow» or «it is winter» loop

max := «the largest number in the list»

d := «current temperature» + 50

CASE color FROM «the darkest» to «the lightest» OF

put («the person’s full name»);
```

Table 3–2 shows the commands and key bindings for manipulating placeholders.

Table 3–2 Commands and Key Bindings for Placeholders

Description	Key Binding	Portable Command
Expand a token.	Ctrl/E	EXPAND
Collapse a token.	PF1-Ctrl/E	COLLAPSE
Go to next placeholder.	Ctrl/N	NEXT PLACEHOLDER
Go to previous placeholder.	Ctrl/P	PREVIOUS PLACEHOLDER
Delete a placeholder.	Ctrl/K	DELETE PLACEHOLDER
Restore a deleted placeholder.	PF1-Ctrl/K	RESTORE PLACEHOLDER
Enter pseudocode.[1]	PF1-spacebar	ENTER PSEUDOCODE
Enter a comment (block).[1]	PF1-B	ENTER COMMENT BLOCK
Enter a comment (line).[1]	PF1-L	ENTER COMMENT LINE

[1]See Section 3.3 for information about pseudocode programming.

3.2.1 Expanding a Token or Placeholder

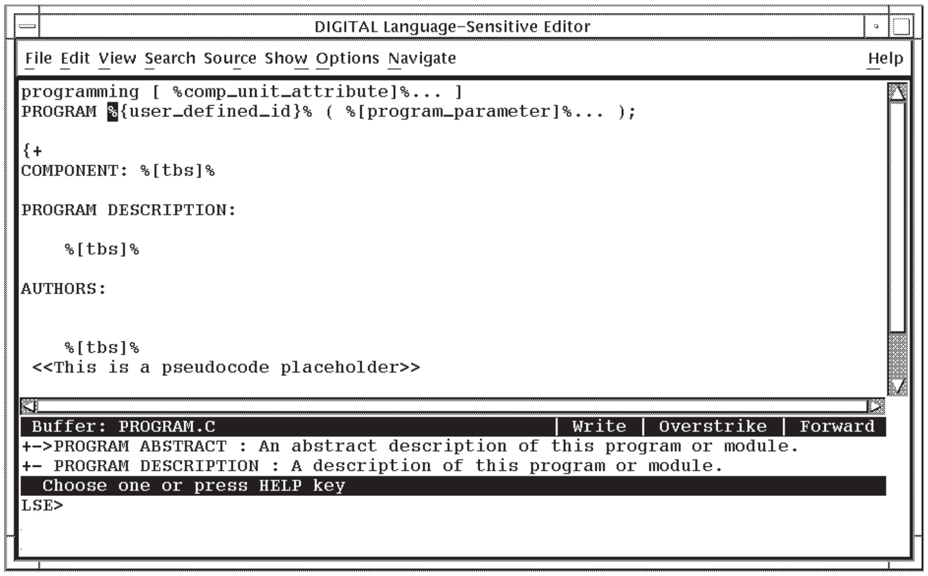
To expand a token or placeholder, place the text cursor on the token or placeholder and use the EXPAND command (or press Ctrl/E). Placeholders help you supply the appropriate syntax in a given context.

You do not have to type an entire token name with LSE. You can abbreviate token names, usually by supplying the first three or four characters. If there are other tokens with the same initial string of characters, LSE provides you with a menu to select the token you want when you press Ctrl/E to expand the token. LSE will not provide you with a menu if the abbreviation is the actual name of another token.

You can type a token name anywhere in the editing buffer and press Ctrl/E to produce the template for that token. A token name does not have to be typed over a placeholder.

Figure 3–2 shows how to expand a text string and select a token (PROGRAM) from the list that is displayed. In this case, the cursor is positioned over the first character of the placeholder `%{user_defined_id}%` and CTRL/E has been pressed. The menu has appeared at the bottom of the screen, allowing you to select either PROGRAM ABSTRACT or PROGRAM DESCRIPTION. Depending on your selection, the appropriate template will be inserted into the buffer at the current cursor position, immediately preceding (and displacing to a later location) the placeholder.

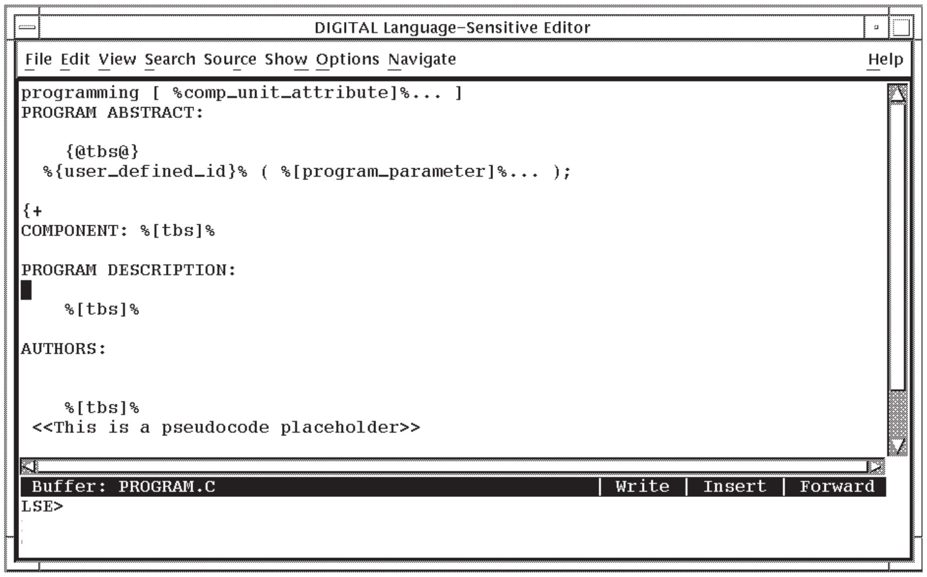
Figure 3–2 Expanding a Token


DIGITAL Language-Sensitive Editor
File Edit View Search Source Show Options Navigate Help

```
programming [ %comp_unit_attribute]%... ]
PROGRAM %{user_defined_id}% ( %[program_parameter]%... );

{+
COMPONENT: %[tbs]%

PROGRAM DESCRIPTION:

    %[tbs]%

AUTHORS:


    %[tbs]%
 <<This is a pseudocode placeholder>>
 Buffer: PROGRAM.C                                | Write | Overstrike | Forward
+->PROGRAM ABSTRACT : An abstract description of this program or module.
+- PROGRAM DESCRIPTION : A description of this program or module.
  Choose one or press HELP key
LSE>
```

Figure 3–3 shows the effect of choosing PROGRAM ABSTRACT in Figure 3–2 from the list.

Figure 3–3 Sample Expanded Token


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Navigate
Help

```
programming [ %comp_unit_attribute]%... ]
PROGRAM ABSTRACT:

    {@tbs@}
  %{user_defined_id}% ( %[program_parameter]%... );

{+
COMPONENT: %[tbs]%

PROGRAM DESCRIPTION:

    %[tbs]%

AUTHORS:


    %[tbs]%
 <<This is a pseudocode placeholder>>
```

```
 Buffer: PROGRAM.C                                 | Write | Insert | Forward
LSE>
```

If you type over a placeholder, it is automatically removed and the text you type is inserted in its place.

LSE initially provides a template containing language-specific placeholders when you create a new file using the language specified by the file name's extension. For example, a file created with a .C extension results in LSE placing initial placeholders for the C language into the empty editing buffer.

The following example shows required and optional placeholders. Consider the first placeholder in the following example; when expanded, it becomes the second placeholder.

```
INTEGER {identifier}...

INTEGER id,
       [identifier]...
```

The first appearance of the identifier placeholder is surrounded by braces because you must supply at least one identifier in this declaration. The second appearance is surrounded by brackets because additional identifiers are optional.

3.2.2 Types of Placeholders

Table 3–3 describes the types of LSE placeholders.

Table 3–3 Types of LSE Placeholders

Placeholder Type	Description
Non-terminal placeholder	When expanded, inserts a programming language template or additional language constructs.
Terminal placeholder	When expanded, supplies a description in another window of what needs to be typed in place of the placeholder.
Menu placeholder	When expanded, provides you with options that can be selected and expanded into templates. When you expand a menu placeholder, you can move through the options by using the arrow keys. To select an option, press Ctrl/E, Return, or Enter. To exit from the menu without selecting an option, press the space bar.
List placeholder	When expanded, is automatically duplicated. Generally, these placeholders represent items such as identifiers, statements, or expressions.
Pseudocode placeholder	Contains comments about design information. Unlike regular placeholders, pseudocode placeholders are obtained by using the ENTER PSEUDOCODE command (or pressing PF1-spacebar); LSE inserts pseudocode placeholder delimiters into the editing buffer within which you can enter your design information. You can move pseudocode placeholder text into program comments.

You can construct a complete program by repeatedly expanding placeholders, but you do not have to continuously expand placeholders until you reach a terminal placeholder. Instead, you might find it more appropriate to type in the desired program text yourself at a higher level.

3.2.3 Indentation Control

LSE controls the indentation of tokens and placeholders placed in an editing buffer with the EXPAND command. Tabulation is determined by the format of the placeholder or token template. In addition, LSE provides the SET BUFFER TAB INCREMENT command, which enables you to adjust indentation to your needs.

When LSE expands a template, it inspects each line of the template for leading tab characters. Each tab character is evaluated using the current tab-increment setting. Thus, if the tab increment is three, a leading tab is expanded into three spaces. The initial tab increment for a buffer is taken from the language definition. You can change the tab increment setting with the SET BUFFER TAB INCREMENT command.

The starting column for the expansion is determined by the first line of the template. If the first line is null (an empty string), the expansion starts under the first nonblank character in the line containing the item being expanded. If the line is not empty, the expansion starts in the first character position of the item being expanded.

For example, suppose you have a placeholder with <TAB> indicating a single ASCII tab character defined, as follows:

```
PLSE NEW PLACEHOLDER RECORD_TYPE_DEFINITION NONTERMINAL
      . . .
PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD RECORD_TYPE_DEFINITION
PLSE SET PLACEHOLDER BODY LINE "TABrecord" -
 EXPAND 0 SPACE NEXT ADD RECORD_TYPE_DEFINITION
PLSE SET PLACEHOLDER BODY LINE "TAB TAB[component declaration]..." -
 EXPAND 0 SPACE NEXT ADD RECORD_TYPE_DEFINITION
PLSE SET PLACEHOLDER BODY LINE "TAB TAB[variant_part]" -
 EXPAND 0 SPACE NEXT ADD RECORD_TYPE_DEFINITION
PLSE SET PLACEHOLDER BODY LINE "TABend record" -
 EXPAND 0 SPACE NEXT ADD RECORD_TYPE_DEFINITION
```

You have the following text:

```
  type MYTYPE is {record_type_definition};
```

If you expand the placeholder with the tab increment set to 4, the result is as follows:

```
  type MYTYPE is
      record
          [component declaration]...
          [variant_part]
      end record
```

The third and fourth lines each have a single leading tab and no leading spaces, whereas the second and fifth lines each have four leading spaces. This is because LSE compresses spaces into tabs after positioning the expanded text.

Using the same example text, remove the first empty line in the placeholder definition and the first tab from the remaining lines, as follows:

```
PLSE NEW PLACEHOLDER RECORD_TYPE_DEFINITION NONTERMINAL
      . . .
PLSE SET PLACEHOLDER BODY LINE "record" -
 EXPAND 0 SPACE NEXT ADD RECORD_TYPE_DEFINITION
PLSE SET PLACEHOLDER BODY LINE "TAB[component declaration]..." -
 EXPAND 0 SPACE NEXT ADD RECORD_TYPE_DEFINITION
PLSE SET PLACEHOLDER BODY LINE "TAB[variant_part]" -
 EXPAND 0 SPACE NEXT ADD RECORD_TYPE_DEFINITION
PLSE SET PLACEHOLDER BODY LINE "end record" -
 EXPAND 0 SPACE NEXT ADD RECORD_TYPE_DEFINITION
```

Then, if you expand the placeholder with the tab increment set to 4, the result is as follows:

```
  type MYTYPE is record
                     [component_declaration]...
                     [variant_part]
                 end record
```

Insert a line containing a single blank at the front of the definition, as follows:

```
PLSE NEW PLACEHOLDER RECORD_TYPE_DEFINITION NONTERMINAL
      . . .
PLSE SET PLACEHOLDER BODY LINE " " -
 EXPAND 0 SPACE NEXT ADD RECORD_TYPE_DEFINITION
PLSE SET PLACEHOLDER BODY LINE "record" -
 EXPAND 0 SPACE NEXT ADD RECORD_TYPE_DEFINITION
PLSE SET PLACEHOLDER BODY LINE "TAB[component declaration]..." -
 EXPAND 0 SPACE NEXT ADD RECORD_TYPE_DEFINITION
PLSE SET PLACEHOLDER BODY LINE "TAB[variant_part]" -
 EXPAND 0 SPACE NEXT ADD RECORD_TYPE_DEFINITION
PLSE SET PLACEHOLDER BODY LINE "end record" -
 EXPAND 0 SPACE NEXT ADD RECORD_TYPE_DEFINITION
```

If you expand the placeholder with the tab increment set to 4, the result is as follows:

```
  type MYTYPE is
                 record
                     [component_declaration]...
                     [variant_part]
                 end record
```

3.3 Creating Pseudocode

Pseudocode lets you sketch your design ideas. Pseudocode placeholders are placeholders enclosed in double angle brackets («») containing natural-language text that expresses design information. They can be customized by using the SET LANGUAGE PSEUDOCODE DELIMIT command.

You can mix pseudocode and final source code, as shown below. Enter pseudocode by issuing the ENTER PSEUDOCODE (PF1 Spacebar) command, then type your pseudocode text. For example:

```
#include <stdlib>
typedef int integer_matrix[10][10];
integer_matrix *matrix_multiply (integer_matrix *left, integer_matrix *right);

/*
**++
**  FUNCTIONAL DESCRIPTION:
**
**      This function computes the matrix product of two integer matrices.
**
**      It uses a simple, triple-nested loop, and does not do any checking to
**      see if the matrices conform.
**
**  FORMAL PARAMETERS:
**
**      left:
**          The left operand.
**
**      right:
**          The right operand.
**
**  RETURN VALUE:
**
**      The result of multiplying the two matrices.
**
**--
*/
integer_matrix *matrix_multiply (integer_matrix *left, integer_matrix *right)
{
    integer_matrix result_matrix; 1

    «Loop over the rows of the left matrix» 2
    return result_matrix; 3
}
```

The callouts in the previous example denote the following:

1 Shows final source code

2 Uses pseudocode as a loop statement

3 Shows final source code

To use pseudocode, pseudocode placeholder delimiters must be defined for the target language. You can use a compiler to process pseudocode to detect syntax errors and generate SCA data. See Section 3.1 for information about compiling a program with LSE.

3.3.1 Moving Pseudocode to Comments

LSE allows you to preserve the descriptive pseudocode text for later use as design information. To change pseudocode to comment lines issue the ENTER COMMENT command. For example, if you apply the ENTER COMMENT (PF1 B) command to the pseudocode line shown in the example code in Section 3.3, the result is as follows:

```
/*
**  Loop over the rows of the left matrix
*/
{@tbs@}
return result_matrix;
```

You can now fill in the next level of detail of your design, as follows:

```
/*
**  Loop over the rows of the left matrix
*/
for (i = 1;  i < «matrix size»;  i++)
{
    «Loop over the columns of the right matrix» 1
};
return result_matrix;
```

Note that 1 shows a new pseudocode line.

If you place the cursor in an existing comment and press PF1-L or PF1-B, LSE converts the next pseudocode placeholder into a comment. If the cursor is not in a comment or on a placeholder when you press PF1-L or PF1-B, LSE inserts a new comment and puts the cursor on the first placeholder after the beginning of the comment.

3.3.2 Processing Pseudocode

When you compile pseudocode, the compiler checks for syntax errors and generates analysis (.ANA) files. SCA uses these files to generate reports about pseudocode, including call trees, dependency tables, and other cross-referencing. SCA provides query support for pseudocode just as it does for source code.

To compile your unfinished programs with LSE, choose Compile Buffer from the Source menu. If your program has errors in it, choose Review Buffer from the Source menu and see what syntax errors you have.

You might want to load the .ANA file into the SCA and generate a Program Design Facility (PDF) report. See the *Source Code Analyzer Command-Line Interface and Callable Routines Reference Manual* and the *Guide to Detailed Program Design for OpenVMS Systems* for more information.

3.4 Formatting Comments

LSE recognizes many of the comments that occur in code. In many cases, LSE handles comments specially to help you keep comments aligned. You can use the ALIGN command to align all the comments within a region so they line up in the same column. You can use the FILL command to both align comments and fill out each comment line by putting as many words on a line as will fit within the margins. In addition, LSE treats comments specially when you erase or duplicate a placeholder.

Special handling of comments applies only to trailing comments. A trailing comment is one that occurs as the last item on a line, excluding blank space. Lines containing only comments are considered to be trailing comments.

Two types of comments are recognized: bracketed comments and line comments. A **bracketed comment** requires both a beginning and ending delimiter. For example, Pascal uses either braces ({}) or parentheses and asterisks ((**)) as bracketed comment delimiters. A **line comment** begins with a comment delimiter, but is terminated by the end of the line. For example, Ada uses a double dash (--) to introduce a line comment, whereas HP Fortran uses an exclamation mark (!).

Note that some languages, such as BLISS, have both bracketed and line comments. The ALIGN command aligns all trailing comments within the current selected region so they start in the same column. The default behavior is to align the comments under the first comment within the region. You can also specify an explicit column with the /COMMENT_COLUMN qualifier.

In the following examples, /COMMENT_COLUMN=CONTEXT_DEPENDENT is in effect. An example of the ALIGN command is as follows:

```
IF (col >= R_Margin) THEN ! This is the start of an
   BEGIN                    ! extended end-of-line comment block
   i := i + l ;
   j := j + i ;  ! another comment
 !to be filled
```

When aligned, it would look like this:

```
IF (col >= R_Margin) THEN ! This is the start of an
   BEGIN                  ! extended end-of-line comment block
   i := i + l ;
   j := j + i ;  ! another comment
                 ! to be filled
```

The operation of the FILL command depends on the FILL setting of the current language. If the FILL setting is TEXT, the FILL command performs an ordinary FILL operation. This is useful for text files or for files written in a markup language, such as HP Standard Runoff, but is usually inappropriate for most programming languages. If the FILL setting is COMMENTS, the FILL command affects only the text within the trailing comments of the region. As with the ALIGN command, you can use the /COMMENT_COLUMN qualifier to explicitly specify the column in which to align the comments.

An example of the FILL command is as follows:

```
IF (col >= R_Margin) THEN ! This is the start of an
   BEGIN                    ! extended end-of-line comment block
   i := i + l ;
   j := j + i ;  ! another comment
 !to be filled
```

When filled, it would look like this:

```
IF (col >= R_Margin) THEN ! This is the start of an extended
   BEGIN                  ! end-of-line comment block
   i := i + l ;
   j := j + i ;           ! another comment to be filled
```

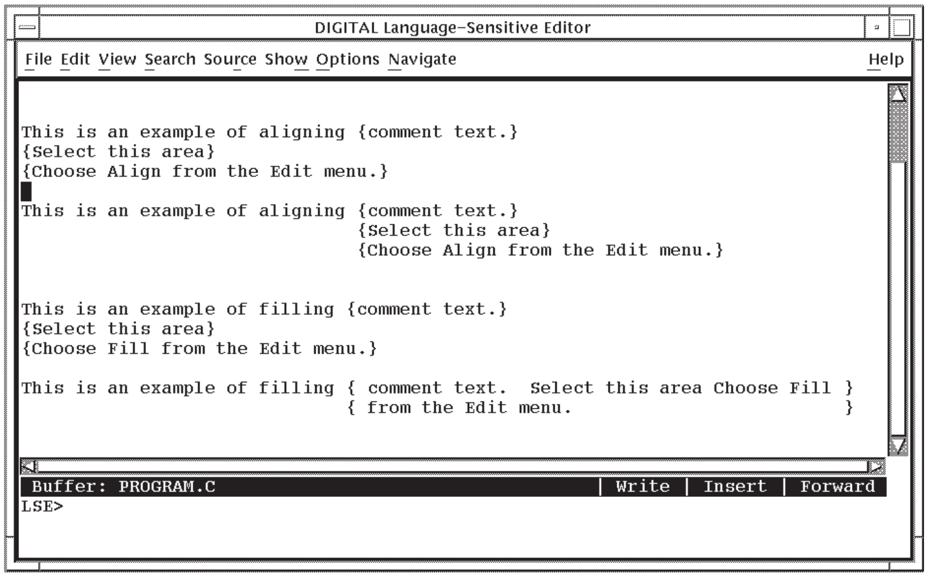
Note that LSE deleted a comment delimiter. LSE inserts or deletes comment delimiters as necessary when you expand or erase placeholders inside comments.

Comments at the end of a line are called **trailing comments**. Lines that contain only comments are also considered trailing comments. You can automatically format trailing comment-line boundaries without changing the format of the programming code by choosing Align or Fill from the Edit menu.

Aligning comment areas left-justifies comments in a selected area, but maintains the comments as they were entered. Filling comment areas also left-justifies comments, but fills the comment line out to the right margin and reformats the comments; comment text is maintained. LSE aligns and fills comments under the first comment in the selected area.

To align comment areas, select the text and choose Align from the Edit menu. To fill comment areas, select the text and choose Fill from the Edit menu. Figure 3–4 shows the effect of aligning and filling comments.

Figure 3–4 Aligning and Filling Comments


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Navigate
Help

```
This is an example of aligning {comment text.}
{Select this area}
{Choose Align from the Edit menu.}

This is an example of aligning {comment text.}
                               {Select this area}
                               {Choose Align from the Edit menu.}


This is an example of filling {comment text.}
{Select this area}
{Choose Fill from the Edit menu.}

This is an example of filling { comment text.  Select this area Choose Fill }
                              { from the Edit menu.                         }

 Buffer: PROGRAM.C                                     | Write | Insert | Forward
LSE>
```

LSE inserts comment delimiters as necessary when you expand placeholders inside comments. LSE will delete comment delimiters as necessary when you erase placeholders.

3.5 Outlining Programming Code

LSE generates overview lines to outline programming code. An **overview line** represents a sequence of lines in a buffer. An overview line is displayed as a pseudocode placeholder with an ellipsis (. . .) inside it. For example:

```
«--Put each pair of numbers in order...»
```

You can collapse a programming language construct into an overview line based on its indentation value. LSE determines the relative level of a line in a program by comparing the indentation of programming language constructs with the indentation of other constructs. LSE has two kinds of indentation: **visible indentation** and **adjusted indentation**.

The visible indentation is the column number of the first visible character on the line. The visible indentation of a blank line is 0.

The adjusted indentation is a defined value you give to assign relative position to program constructs so they can be outlined. You cannot see adjusted indentation because LSE does not move any text; LSE treats lines as though the indentation has changed.

Details of the overviews are specific to the language you are using. Sometimes visible indentation information alone is not enough to generate useful overviews.

By collapsing levels of programming code into overview lines, you can get an overview of a program's outline. You can move the text cursor through the program outline and expand the overview lines to expose the underlying programming code.

Table 3–4 shows the commands for outlining and expanding programming language constructs.

Table 3–4 Collapsing and Expanding Programming Language Constructs

Menu Command	Description
Overview Source	Shows a top-level overview of the buffer.
View Source	Shows the source code in the buffer.
Focus	Replaces all programming language constructs above and below the current construct to an overview line. Focus also collapses all lines within the current programming language construct that are at a greater level of indentation than the construct under the cursor to overview lines.
Expand	Replaces an overview line with the next level of detail for the underlying source code.
Expand All	Replaces an overview line with the actual source code it conceals (the lowest level of detail).
Collapse	Replaces a sequence of source code lines with an overview line.
Collapse All	Replaces all programming language constructs with overview lines.

The grouping of lines depends on indentation conventions, combined with simple language-specific syntax processing. In the following example, the first five lines constitute a group:

```
WHILE x
    BEGIN
    a = b
    c = d
    END
q = r
```

When the group is compressed, a single line represents the five lines, as follows:

```
«WHILE x...»
q = r
```

You can nest groups of lines so you can present a single program at various levels of detail. The following example shows an indented, nested group of lines associated with an if statement:

```
--Put each pair of numbers in order
for J in 1 .. HOW_MANY - 1 loop
    if NUMBERS(J) > NUMBERS(J+1) then
        --Interchange the numbers
        TEMP := NUMBERS(J);
        NUMBERS(J) := NUMBERS(J+1);
        NUMBERS(J+1) := TEMP;

        --We are not finished sorting
        SORTED := FALSE;
    end if;
end loop;
```

The indented lines can be represented by overview lines, as follows:

```
--Put each pair of numbers in order
for J in 1 .. HOW_MANY - 1 loop
    if NUMBERS(J) > NUMBERS(J+1) then
        « --Interchange the numbers...»
        « --We are not finished sorting...»
    end if;
end loop;
```

The entire code fragment can be represented by a single overview line, as follows:

```
« --Put each pair of numbers in order...»
```

3.5.1 Debugging Adjustment Definitions

There are two ways to debug SET ADJUSTMENT definitions. One way is to use the viewing commands on sample text and experiment with SET ADJUSTMENT definitions until you are satisfied with the results.

Alternatively, you can debug your definitions by using the VIEW SOURCE DEBUG command. This command generates a representation of the source buffer, indented as LSE perceives the indentation, as specified by the adjustment definitions. The result is displayed in a system buffer, $OVERVIEW. For example:

```
NEW ADJUSTMENT "!"
SET ADJUSTMENT CURRENT -1
SET ADJUSTMENT UNIT ON

NEW ADJUSTMENT BEGIN
SET ADJUSTMENT SUBSEQUENT 2
SET ADJUSTMENT COMPRESS OFF
```

```
NEW ADJUSTMENT END
SET ADJUSTMENT CURRENT -1
SET ADJUSTMENT SUBSEQUENT -2
```

Suppose you have executed the VIEW SOURCE DEBUG command when the cursor is in a buffer containing the following source text (assuming each line begins in column 1):

```
    ! Swap the numbers
    BEGIN
    temp = a;
    a = b;
    ! Get the value from temporary storage.
    b = temp
    END
```

This puts the following text in the $OVERVIEW buffer:

```
U          3  0  0 ! Swap the numbers.
 NC        4  0  1  BEGIN
           1  2  3    temp = a;
           1  2  3    a = b;
U          3  2  2   ! Get the value from temporary storage.
           1  2  3    b = temp
           5  2  2   END
--------------------------------------------------------------------------
    Left margin numbers (in order):
        adjustment number
        /SUBSEQUENT running total
        effective indentation

    U = /UNIT  NC = /NOCOMPRESS  NO = /NOOVERVIEW  N# = /NOCOUNT
    IP = /INHERIT=PREV  IN = /INHERIT=NEXT, I0 = /INHERIT=NEITHER
    I< = /INHERIT=MINIMUM,  I> = /INHERIT=MAXIMUM

   Adjustment
  number:   name                                    use count

       1: default                                       3
       2: default blank                                 0
       3: !                                             2
       4: BEGIN                                         1
       5: END                                           1
```

The information in the $OVERVIEW buffer explains the following:

- The first line "`U          3  0  0  ! Swap the numbers.`"; the adjustment for “!” used adjustment 3. This adjustment includes a UNIT ON value. This line’s effective indentation is 0 because its visible indentation is 1 and its adjustment includes CURRENT –1 value.

- The second line used adjustment 4 (the adjustment for BEGIN). This adjustment includes a COMPRESS OFF value. This line's effective indentation is 1. The adjustment for this line also includes SUBSEQUENT 2, so the subsequent running total for the next line is 2.
- The third line used the default adjustment (adjustment 1) because its text does not match the pattern of any defined adjustment. This line's effective indentation is 3 because its visible indentation is 1 and the running total of subsequent values is 2.
- The fourth and sixth lines are similar to the third line.
- The fifth line used the same adjustment as the first line. Because the running total of subsequent values is 2 when you reach this line, its effective indentation is 2.
- The seventh line used adjustment 5 (the adjustment for END). This line's effective indentation is 2, and its visible indentation is 1. This is adjusted by the subsequent running total of 2, and the adjustment keyword CURRENT –1. The calculation is: 1 + 2 –1 = 2.

You can judge the quality of your definitions by looking at the relative indentation of the lines. A line hides all the lines that follow it until a line occurs that has the same or less indentation. In this example, the first comment hides all the other lines; the second comment hides only one line; and none of the assignment lines hide any lines.

When some lines have an adjusted indentation that is negative, the display is appropriately altered.

3.5.2 Special Processing for FORTRAN

In the FORTRAN language, the adjustment actions LSE applies depend on whether a line is a comment line, a continuation line, or neither of these.

A continuation line is treated as though defined with the following commands:

```
SET ADJUSTMENT INHERIT MAXIMUM
SET ADJUSTMENT CURRENT 1
SET ADJUSTMENT UNIT ON
```

If the line is a comment line, LSE matches against adjustments that contain the FORTRAN_COMMENT pattern element (described later in this section). If the line is neither a comment nor a continuation line, LSE matches against adjustments for noncomments (adjustments that do not contain the FORTRAN_COMMENT pattern element) and examines only the text in the statement field. CURRENT 1 refers to the first column in the statement field.

Pattern elements defined for FORTRAN are as follows:

FORTRAN_FUNCTION

LSE defines a special pattern, called FORTRAN_FUNCTION, to match the first line of a FORTRAN function subprogram. The format of such a line is as follows:

type [*number] FUNCTION

type
Is one of the following keywords:

BYTE	DOUBLE COMPLEX	LOGICAL
CHARACTER	DOUBLE PRECISION	REAL
COMPLEX	INTEGER	

number
Is either a series of digits or an asterisk (*).

FORTRAN_COMMENT

LSE recognizes a special pattern element, FORTRAN_COMMENT. This pattern element can appear only as the first element of a pattern and can apply the definition only if the source line is known to be a comment. This is conceptually similar to CURRENT, which applies the definition if the text starts in the correct column.

An adjustment is a comment adjustment if the pattern begins with FORTRAN_COMMENT. Otherwise, it is a noncomment adjustment.

Examples

Each of the following examples starts in column 1. All examples are for ANSI format.

1. The following definition matches each of the subsequent lines of code:

   ```
   SET ADJUSTMENT "$(FORTRAN_COMMENT)C**"
   ```

   ```
   C**
   c**
   C**stuff
   ```

   However, the definition does not match any of the following lines:

```
C stuff
C **
CC**
    C**
***
!**
12345 x = y
           end
     1continuation
```

2. The following definition matches each of the subsequent lines of code:

   ```
   SET ADJUSTMENT "$(FORTRAN_COMMENT)!"


   !stuff
   !
   !*
   ```

   However, it does not match either of the following lines:

   ```
   C
       !
   ```

3. The following definition matches each of the subsequent lines of code:

   ```
   SET ADJUSTMENT "endif"


   12345 endif
         endif
        0endif
   52            endif
                  endif
   ```

   However, it does not match either of the following lines:

   ```
   C endif
        1endif
   ```

4. The following definition matches each of the subsequent lines of code:

   ```
   SET ADJUSTMENT "$(COLUMN=1)end"


   12345 end
         end
        0end
   ```

However, it does not match any of the following lines:

```
12345      end
           end
C end
```

4
Defining Languages for LSE

This chapter describes how to define your own templates and how to create and use environment files. It also includes two examples of environment files created with LSE. The following information is presented:

- Defining a text template for a programming language
- Defining a programming language
- Defining adjusted indentation
- Saving language definitions
- Using aliases
- Using LSE packages

LSE provides a mechanism for interfacing non-HP processors to the diagnostic review facility.

4.1 Defining a Text Template

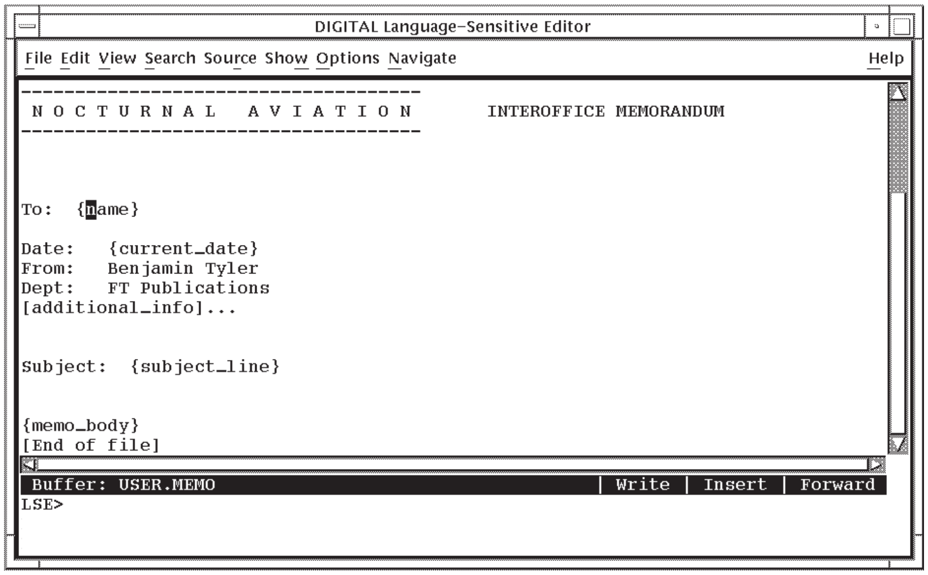
When defining a language, you create a file in which to put LSE commands that will be used to define elements of your language. Once the file is complete, you execute the commands in the source file and create an environment file for use with LSE. The following section describes how to create a language for writing memos. This memo language is defined using LSE language, placeholder, and token definitions.

Before starting, note the following coding rules:

- If a command and its keywords extend beyond one line, a hyphen (`-`) must be the last character on each continuation line, except the last line of the command.
- Comment lines begin with an exclamation point (`!`) and ignore leading blank spaces.
- Lines containing only comments do not terminate continued commands.

Figure 4–1 shows a screen display of the memo template as it appears after you expand the initial string, `{memo_template}`.

Figure 4–1 Memo Template


DIGITAL Language-Sensitive Editor
File Edit View Search Source Show Options Navigate
Help

```
-------------------------------------
 N O C T U R N A L   A V I A T I O N      INTEROFFICE MEMORANDUM
-------------------------------------



To:  {name}

Date:   {current_date}
From:   Benjamin Tyler
Dept:   FT Publications
[additional_info]...


Subject:  {subject_line}


{memo_body}
[End of file]
 Buffer: USER.MEMO                                    | Write | Insert | Forward
LSE>
```

4.1.1 Language Definition

All template definitions, whether text- or language-oriented, must begin with a language definition. To define a language, use the LSE command NEW LANGUAGE. This command takes keywords that set the characteristics of the language.

The following examples show the Portable language definition commands for a memo:

```
PLSE NEW LANGUAGE memo
    PLSE SET LANGUAGE FILE TYPES ".MEMO .FOO" ADD memo
    PLSE SET LANGUAGE INITIAL STRING "{memo_template}" memo
    PLSE SET LANGUAGE REQUIRED DELIMIT "{" "}" memo
    PLSE SET LANGUAGE REQUIRED LIST DELIMIT "{" "}..." memo
    PLSE SET LANGUAGE OPTIONAL DELIMIT "[" "]" memo
    PLSE SET LANGUAGE OPTIONAL LIST DELIMIT "[" "]..." memo
    PLSE SET LANGUAGE IDENTIFIER CHARACTERS -
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890" memo
    PLSE SET LANGUAGE PUNCTUATION CHARACTERS ",;()" memo
    PLSE SET LANGUAGE TAB INCREMENT 4 memo
```

The IDENTIFIER CHARACTERS keywords determine what sequences of characters LSE considers to be a word. A word is any sequence of identifier characters delimited by either blank space or a nonidentifier character. The IDENTIFIER CHARACTERS keywords also specify what characters can appear in alias names. Typically, identifier characters include the alphabet (both uppercase and lowercase), digits, and a few special characters.

The INITIAL STRING keywords specify the initial text that appears in a newly created file. LSE automatically inserts this text whenever you use either the LSEDIT command from the command line or the NEW FILE command from within LSE to create a new file.

The FILE TYPES keywords specify what file types correspond to this language. In this case, when you edit a file with the file type .MEMO or .FOO, LSE automatically sets the language MEMO for the buffer.

The TAB INCREMENT keywords set the tab stops at every four columns.

The DELIMIT keywords specify the starting and ending strings that delimit placeholders. Placeholders can be required or optional, and they can specify single constructs or lists of constructs. There are four types of placeholder delimiters: required, optional, required list, and optional list. The delimiters for each type are specified as a pair of quoted strings and parentheses. By convention, list placeholders are indicated by appending ellipses (. . .) to the corresponding nonlist placeholders.

The PUNCTUATION CHARACTERS keywords specify the characters that are considered punctuation marks or delimiters. Punctuation characters are not important to the MEMO language. The value specified in the language definition is the default value. See Section 4.2 for a more detailed description of both punctuation characters and identifier characters.

4.1.2 Placeholder Definitions

There are three types of placeholders:

- Nonterminal placeholders, which expand into text that is inserted into the buffer
- Terminal placeholders, which expand into descriptive text that is displayed in a temporary window
- Menu placeholders, which provide a list of options for expanding the placeholder

To define a placeholder, use the NEW PLACEHOLDER command. For the MEMO language example, define the initial string "memo_template" as a nonterminal placeholder. In this case, the definition will contain a **placeholder body**. A placeholder body contains the text that is inserted in the buffer when you expand the placeholder. Each line of the placeholder body must be enclosed in quotation marks. Quotation marks with no text produce a blank line when expanded.

The following example shows how the "memo_template" placeholder is defined:

```
PLSE NEW PLACEHOLDER "memo_template" NONTERMINAL memo
    PLSE SET PLACEHOLDER AUTO SUBSTITUTE OFF "memo_template" memo
    PLSE SET PLACEHOLDER DESCRIPTION "Interoffice Memorandum" "memo_template" memo
    PLSE SET PLACEHOLDER DUPLICATION CONTEXT_DEPENDENT "memo_template" memo
    PLSE SET PLACEHOLDER PSEUDOCODE ON "memo_template" memo
    PLSE SET PLACEHOLDER BODY LINE "-------------------------------------" EXPAND 0 SPACE
NEXT ADD "memo_template" memo
    PLSE SET PLACEHOLDER BODY LINE " N O C T U R N A L   A V I A T I O N    INTEROFFICE
MEMORANDUM" EXPAND 0 SPACE NEXT ADD "memo_template" memo
    PLSE SET PLACEHOLDER BODY LINE "-------------------------------------" EXPAND 0 SPACE
NEXT ADD "memo_template" memo
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD "memo_template" memo
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD "memo_template" memo
    PLSE SET PLACEHOLDER BODY LINE "To: {name}" EXPAND 0 SPACE NEXT ADD "memo_template" memo
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD "memo_template" memo
    PLSE SET PLACEHOLDER BODY LINE "Date:  {current_date}" EXPAND 0 SPACE NEXT ADD
"memo_template" memo
    PLSE SET PLACEHOLDER BODY LINE "From:  Benjamin Tyler" EXPAND 0 SPACE NEXT ADD "memo_template"
memo
    PLSE SET PLACEHOLDER BODY LINE "Dept:  FT Publications" EXPAND 0 SPACE NEXT ADD
"memo_template" memo
    PLSE SET PLACEHOLDER BODY LINE "[additional_info]..." EXPAND 0 SPACE NEXT ADD
"memo_template" memo
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD "memo_template" memo
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD "memo_template" memo
    PLSE SET PLACEHOLDER BODY LINE "Subject: {subject_line}" EXPAND 0 SPACE NEXT ADD
"memo_template" memo
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD "memo_template" memo
    PLSE SET PLACEHOLDER BODY LINE "{memo_body}" EXPAND 0 SPACE NEXT ADD "memo_template" memo
```

The DESCRIPTION keyword specifies a single line of text displayed along with the placeholder name whenever the placeholder appears in a menu.

The text following the DESCRIPTION keyword in the "memo_template" placeholder definition is the placeholder body. Placeholder bodies represent the text that is inserted into the buffer when the placeholder is expanded. Text within a placeholder body can include additional placeholders. These placeholders must include the appropriate delimiters so LSE will know how to expand them.

All placeholders found within a placeholder body must be defined. These placeholders do not have to be defined in any particular order. When defining placeholders, do not include delimiters.

In this case, you have defined the `{name}`, `{current_date}`, `{subject_line}`, and `{memo_body}` placeholders as terminal placeholders, specified by the TERMINAL keyword. Terminal placeholders, when expanded, provide a description of the appropriate values that you must type over the text to replace the placeholder. The text for a terminal placeholder can be as many lines as needed to explain the correct text insertion for the placeholder. You must use quotation marks at the beginning and end of each line of text.

The `[additional_info]` placeholder appears within the delimiters for an optional list. See the `[additional_info]` placeholder definition at the end of this section for more information on list placeholders.

The following example shows how the terminal placeholders are defined:

```
PLSE NEW PLACEHOLDER name TERMINAL memo
    PLSE SET PLACEHOLDER TERMINAL LINE "Name of the person receiving this memo." -
            NEXT ADD name memo

PLSE NEW PLACEHOLDER current_date TERMINAL memo
    PLSE SET PLACEHOLDER TERMINAL LINE  "Date the memo is written." -
            NEXT ADD current_date memo
    PLSE SET PLACEHOLDER TERMINAL LINE  "Date can be of the format day/month/year." -
            NEXT ADD current_date memo

PLSE NEW PLACEHOLDER subject_line TERMINAL memo
    PLSE SET PLACEHOLDER TERMINAL LINE  "Subject of the memo." -
            NEXT ADD subject_line memo

PLSE NEW PLACEHOLDER memo_body TERMINAL memo
    PLSE SET PLACEHOLDER DESCRIPTION "Body of the memo." memo_body memo
    PLSE SET PLACEHOLDER TERMINAL LINE  "Supply the text of the memo here." -
            NEXT ADD memo_body memo
```

The `"additional_info"` placeholder is defined as a menu placeholder, as follows:

```
PLSE NEW PLACEHOLDER "additional_info" MENU memo
    PLSE SET PLACEHOLDER AUTO SUBSTITUTE OFF "additional_info" memo
    PLSE SET PLACEHOLDER DESCRIPTION "Additional identification information" -
            "additional_info" memo
    PLSE SET PLACEHOLDER DUPLICATION VERTICAL "additional_info" memo
    PLSE SET PLACEHOLDER PSEUDOCODE ON "additional_info" memo
    PLSE SET PLACEHOLDER MENU LINE "Phone" "phone number" TEXT OFF NEXT ADD -
            "additional_info" memo
    PLSE SET PLACEHOLDER MENU LINE "Location" "building location" TEXT OFF -
            NEXT ADD "additional_info" memo
    PLSE SET PLACEHOLDER MENU LINE "Network" "node address" TEXT OFF -
            NEXT ADD "additional_info" memo
    PLSE SET PLACEHOLDER MENU LINE "CC" "carbon copy" TEXT OFF NEXT ADD "additional_info" memo
    PLSE SET PLACEHOLDER MENU LINE "All" "" TEXT OFF NEXT ADD "additional_info" memo
```

Use the MENU keyword to specify that this is a menu placeholder. The placeholder body for a menu placeholder is defined the same way as a nonterminal placeholder and contains the elements of the menu. In this case, each element in the menu is defined as a token. Section 4.1.3 describes how to define tokens.

The `"additional_info"` placeholder is also used as an optional list placeholder, indicated by brackets ([ ]) . . . . Because `"additional_info"` is used as a list placeholder, you need to specify what type of duplication should be performed when the placeholder is expanded by using the DUPLICATION keyword. Duplication options are VERTICAL, HORIZONTAL, and CONTEXT_DEPENDENT. In this case, the VERTICAL option is used, which places the duplicate placeholder on the next line immediately under the original placeholder.

4.1.3 Token Definitions

Tokens are keywords that you type directly into the buffer and expand into additional templates. This example uses tokens for additional fields of the memo header. By using tokens instead of placeholders, you simplify the entry of frequently used options.

To define a token, use the LSE command NEW TOKEN. The LANGUAGE keyword in the following token definitions tells LSE that this token is defined for the MEMO language.

With the DESCRIPTION keyword, you can supply a text string that is displayed with the token name in a menu. In this case, each token definition includes the DESCRIPTION keyword. Thus, when the `"additional_info"` placeholder is expanded into a menu, each token and its description is displayed.

The body of a token is defined in the same way for nonterminal and terminal placeholders.

By defining these elements as tokens, you can type the token name you want directly into the buffer and expand it without going through a menu.

The following example shows how tokens are defined:

```
PLSE NEW TOKEN "All" TERMINAL memo
    PLSE SET TOKEN DESCRIPTION "Choose all options in menu." "All" memo
    PLSE SET TOKEN BODY LINE "Phone" CURRENT 0 SPACE NEXT ADD "All" memo
    PLSE SET TOKEN BODY LINE "Location" CURRENT 0 SPACE NEXT ADD "All" memo
    PLSE SET TOKEN BODY LINE "Network" CURRENT 0 SPACE NEXT ADD "All" memo
    PLSE SET TOKEN BODY LINE "CC" CURRENT 0 SPACE NEXT ADD "All" memo

PLSE NEW TOKEN "CC" TERMINAL memo
    PLSE SET TOKEN DESCRIPTION "Name of person to receive a copy of this memo" CC memo
    PLSE SET TOKEN BODY LINE "CC:     {name}" CURRENT 0 SPACE NEXT ADD CC memo

PLSE NEW TOKEN "Location" TERMINAL memo
    PLSE SET TOKEN DESCRIPTION "Office location" "Location" memo
    PLSE SET TOKEN BODY LINE "LOC:    URE-0096" CURRENT 0 SPACE NEXT ADD "Location" memo

PLSE NEW TOKEN "Network" TERMINAL memo
    PLSE SET TOKEN DESCRIPTION "Network address" "Network" memo
    PLSE SET TOKEN BODY LINE "NET:    EMLEN::SNYDER" CURRENT 0 SPACE NEXT ADD "Network" memo

PLSE NEW TOKEN "Phone" TERMINAL memo
    PLSE SET TOKEN DESCRIPTION "Office phone number" "Phone" memo
    PLSE SET TOKEN BODY LINE "Phone:  523-440-3287" CURRENT 0 SPACE NEXT ADD "Phone" memo
```

The token definitions for `"CC"` and `"All"` include the `{name}` placeholder. All placeholders found within a token body must be defined. In this case, you defined `{name}` in the previous section, so you do not have to define it again.

When the template definition is complete, you must execute your source file. Section 4.4 describes how to execute your source files and create an environment file for later use.

4.2 Defining a Programming Language

This section describes how to define templates for a programming language. The concepts and coding rules for defining templates for a programming language are similar to those introduced in the MEMO language. However, some advanced features necessary for handling more complex templates are introduced.

The following sections explain how the EXAMPLE language was defined. The EXAMPLE language is available on line with LSE.

You should follow a grammar, or other syntax summary, as a guide if you are defining a programming language. A grammar provides a summary from which to work when defining the equivalent LSE definitions. It also helps you avoid mistakes. Example 4–1 contains a syntax summary for the EXAMPLE language.

Those expressions on the left side of the syntax summary are known as nonterminals. Typically, nonterminals with multiple options, such as `statement`, are defined as menu placeholders. Nonterminals that expand into a single option, such as `if_stmnt`, are defined as nonterminal placeholders or tokens. Nonterminals that expand into text you must supply, such as `identifier`, are defined as terminal placeholders. Optional syntactic elements, such as `[ELSE statement_list]`, require placeholders that do not explicitly appear as nonterminals in the syntax summary. Major keywords, such as `IF`, are defined as tokens.

Example 4–1 Syntax Summary for the EXAMPLE Language

```
program_unit        ::= PROCEDURE procedure_name [(parameter_list)] IS
                        [variable_declarations]
                        BEGIN
                        statement_list
                        END
                        ENDPROCEDURE procedure_name
procedure_name      ::= identifier
identifier          ::= alphabetic | alphabetic identifier_chars
identifier_chars    ::= alphanumeric | alphanumeric identifier_chars
alphabetic          ::= a | b | ... | z
numeric             ::= 0 | 1 | ... | 9
alphanumeric        ::= alphabetic | numeric
parameter_list      ::= parameter:type | parameter:type, parameter_list
parameter           ::= identifier
var_decl_list       ::= var_decl; | var_decl; var_decl_list
var_decl            ::= identifier_list : type [init_value_list]
identifier_list     ::= identifier | identifier, identifier_list
init_value_list     ::= initial_value | initial_value, init_value_list
initial_value       ::= boolean_exp | arithmetic_exp
statement_list      ::= statement; | statement; statement_list
statement           ::= if_stmnt | assignment_stmt | loop_construct |exit_stmt
if_stmnt            ::= IF expression THEN statement_list [ELSE statement_list] ENDIF
assignment_stmnt    ::= identifier := expression
expression          ::= boolean_exp | arithmetic_exp
boolean_exp         ::= identifier | unary identifier | identifier relop identifier
relop               ::= = | <> | < | > | <= | >=
arithmetic_exp      ::= identifier oper identifier
oper                ::= + | - | * | /
unary               ::= NOT
loop_construct      ::= [label:] LOOP statement_list END LOOP
label               ::= identifier
exit_stmt           ::= EXIT [label] WHEN expression
type                ::= BOOLEAN | INTEGER
```

4.2.1 Language Definition

To define a language (in this case, EXAMPLE), use the NEW LANGUAGE command and its keywords. The following example shows the language definition for the EXAMPLE language:

```
PLSE NEW LANGUAGE EXAMPLE
  PLSE SET LANGUAGE FILE TYPES ".example" ADD EXAMPLE
  PLSE SET LANGUAGE INITIAL STRING "{program_unit}" EXAMPLE
  PLSE SET LANGUAGE REQUIRED DELIMIT "{" "}" EXAMPLE
  PLSE SET LANGUAGE REQUIRED LIST DELIMIT "{" "}..." EXAMPLE
  PLSE SET LANGUAGE OPTIONAL DELIMIT "[" "]" EXAMPLE
  PLSE SET LANGUAGE OPTIONAL LIST DELIMIT "[" "]..." EXAMPLE
  PLSE SET LANGUAGE PSEUDOCODE DELIMIT "<" ">" EXAMPLE
  PLSE SET LANGUAGE EXPAND CASE AS_IS EXAMPLE
  PLSE SET LANGUAGE DIAGNOSTICS OFF EXAMPLE
  PLSE SET LANGUAGE COMPILE COMMAND "" EXAMPLE
  PLSE SET LANGUAGE BRACKETED COMMENTS "/*" "*/" ADD EXAMPLE
  PLSE SET LANGUAGE TRAILING COMMENTS "--" ADD EXAMPLE
  PLSE SET LANGUAGE IDENTIFIER CHARACTERS -
  "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890$_" EXAMPLE
  PLSE SET LANGUAGE PUNCTUATION CHARACTERS ",;().':" EXAMPLE
  PLSE SET LANGUAGE QUOTES "" EXAMPLE
  PLSE SET LANGUAGE LEFT MARGIN 0 EXAMPLE
  PLSE SET LANGUAGE RIGHT MARGIN 80 EXAMPLE
  PLSE SET LANGUAGE TAB INCREMENT 4 EXAMPLE
  PLSE SET LANGUAGE HELP TOPIC "" EXAMPLE
  PLSE SET LANGUAGE VERSION "2.3" EXAMPLE
  PLSE SET LANGUAGE WRAP OFF EXAMPLE
```

The DIAGNOSTICS keyword specifies whether the compiler for the language creates a diagnostic file when it is compiled from within LSE. If DIAGNOSTICS is specified, LSE automatically appends the DIAGNOSTICS keyword to the COMPILE command. There is no compiler for the EXAMPLE language. This is consistent with the COMPILE command keywords having a null value. If a compiler existed, it could optionally produce a diagnostic file upon compilation. The LSE command REVIEW would then use this file.

The COMMENTS keywords specify the character sequences of comments in the language. The /* and */ keywords specify that bracketed comments begin with the sequence /* and terminate with the sequence */. Bracketed comments can extend over several lines. The trailing comment—specifies that trailing comments begin with a pair of dashes. Trailing comments terminate at the end of the line.

The COMPILE COMMAND keywords specify the default command string to be used to invoke the language processor when you enter the LSE command COMPILE. If there were an actual compiler for the EXAMPLE language, you would specify the command that invokes that compiler here. In the definition of the HP C language, for example, the keywords are COMPILE COMMAND "cc".

The EXPAND CASE keywords specify the case that the text of inserted templates is to have. The AS_IS option specifies that the expanded text is to keep the same case as in the placeholder or token definition.

The FILE TYPES keywords specify what file type corresponds to the EXAMPLE language. When you specify the file type .EXAMPLE, LSE automatically sets the language EXAMPLE for the buffer. You must not specify a file type that is already associated with another language.

The HELP TOPIC keyword specifies a common prefix string to which the `topic_string` parameter specified in a placeholder or token definition is concatenated. For example, suppose a language topic string is "Pascal", and the help library keyword is null. The definition of the token IF contains the keywords HELP TOPIC "statement IF_THEN_ELSE". If a user gives the HELP INDICATED command while positioned on the token IF in a Pascal buffer, LSE forms the string "PASCAL STATEMENT IF_THEN_ELSE" and uses that as an index.

If there is a common prefix for all the help topics for a language, the HELP TOPIC keyword should not be specified. If you have created your own help library for the language, and help for this language is the only topic in this help library, do not specify the HELP TOPIC keyword. By default, LSE looks up the single topic in the library.

The IDENTIFIER CHARACTERS keyword specifies what characters are considered as part of a word. A word in LSE is defined as either any nonblank space, nonidentifier character, or a sequence of identifier characters delimited by blank space or a nonidentifier character. In addition to all alphanumeric characters, you might want to have other characters as identifier characters, such as those that often appear in variable names (for example, $ or _).

The PUNCTUATION CHARACTERS keyword specifies those characters that LSE considers punctuation characters. LSE uses punctuation characters in two ways. Note that both affect how LSE deletes placeholders. The two ways are as follows:

- When a placeholder is erased and only punctuation characters are left on the line, the line is deleted and the punctuation characters are moved to the end of the preceding line. For example:

  ```
  INTEGER ( x,
            [var]);
  ```

When the placeholder `[var]` is erased, only `);` is left on the line. Assuming both `)` and `;` are specified as punctuation characters for the language, `);` is moved to the end of the preceding line. (The comma after `X` is also erased, if specified as the separator for `var`.)

- When a placeholder is erased, LSE examines the characters immediately before and after the placeholder. If neither of these are punctuation characters, LSE inserts a blank between them. For example:

  ```
  IF [NOT]{boolean_expression} THEN
  ```

  If the optional placeholder is deleted and the brace is a punctuation character, the following results:

  ```
  IF {boolean_expression} THEN
  ```

  Thus, in this case it would be better to have the brace not be included as a punctuation character.

The INITIAL STRING keyword specifies the initial text that appears in a newly created buffer. For example, when you create a new file with an extension of .EXAMPLE, the buffer contains just the initial string `{program_unit}`.

The LEFT MARGIN and RIGHT MARGIN keywords specify the left and right margin values, respectively, to be associated with the language. LEFT MARGIN CONTEXT_DEPENDENT means that the left margin varies depending on the surrounding context. For example, the FILL command uses the indentation of the first line of the text being filled as the left margin for the fill operation. When wrap mode is enabled, the right margin value is used to determine where to break the line. It is also used by the FILL command. The FILL command packs text up to, but not beyond, the right margin. When filling comments, the close comment delimiters are placed at the right margin.

The DELIMIT keywords specify the starting and ending strings that delimit placeholders. Placeholders can be required or optional, and they can specify single constructs or lists of constructs. By convention, braces are used for required placeholder delimiters, and square brackets are used for optional delimiters in all HP languages. If these characters are part of the syntax of the language, then some other character is used with them. For example, because braces ({}) have meaning in the C programming language, {@ and @} are used as placeholder delimiters.

The TAB INCREMENT keyword indicates that tab stops are set at every four columns.

The QUOTES keyword specifies the characters to be used in the language as delimiters for strings and escape characters.

For example, SET LANGUAGE QUOTES """’" and SET LANGUAGE ESCAPE "\", which is the specification for some HP languages, specifies that strings in the language can be delimited by either double quotation marks (" ") or single quotation marks (’ ’), and that the escape character for the language is the backslash (\ ). LSE always assumes that the strings begin and end with the same character, but different strings can use different delimiters. The value being provided for the QUOTES keyword must be surrounded by quotation marks; if you want to include the quotation mark character within this value, you must use double quotation marks.

You can optionally use the VERSION keyword to help you keep track of your product. LSE does not actually use this value, except to display it in the SHOW LANGUAGE command. You might want to use it to show what revision number the language definition is, for example, or what version of the associated compiler the templates support.

The WRAP ON keyword specifies whether wrap mode is in effect by default in the language. The WRAP keyword affects the left margin value if the LEFT MARGIN keyword was specified with a value of CONTEXT_DEPENDENT.

4.2.2 Defining Language Elements

You can define the placeholders that can be used in the EXAMPLE language. To define a placeholder, use the NEW PLACEHOLDER command. Referring to the syntax summary, start with the top-level nonterminal of the language, `program_unit`. This is a nonterminal placeholder.

For example:

```
PLSE NEW PLACEHOLDER program_unit NONTERMINAL EXAMPLE
    PLSE SET PLACEHOLDER AUTO SUBSTITUTE OFF program_unit EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "Program unit" program_unit EXAMPLE
    PLSE SET PLACEHOLDER DUPLICATION CONTEXT_DEPENDENT program_unit EXAMPLE
    PLSE SET PLACEHOLDER PSEUDOCODE ON program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "-- [procedure level comments]" EXPAND 0 SPACE NEXT ADD
program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "PROCEDURE {procedure_name} ([parameter list]...) IS"
EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE " [variable_declaration]...;" EXPAND 0 SPACE NEXT ADD
program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "BEGIN" EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE " [statement]...;" EXPAND 0 SPACE NEXT ADD program_unit
EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "END {procedure_name};" EXPAND 0 SPACE NEXT ADD program_unit
EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "-- [procedure level comments]" EXPAND 0 SPACE NEXT ADD
program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "PROCEDURE {procedure_name} ([parameter list]...) IS"
EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "    [variable_declaration]...;" EXPAND 0 SPACE NEXT ADD
program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "BEGIN" EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "    [statement]...;" EXPAND 0 SPACE NEXT ADD program_unit
EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "END {procedure_name};" EXPAND 0 SPACE NEXT ADD program_unit
EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "" EXPAND 0 SPACE NEXT ADD program_unit EXAMPLE
```

The NONTERMINAL keyword means that the body for the placeholder follows. The placeholder body for `program_unit` is specified after the keywords. A placeholder body is the template inserted into the buffer when you expand `program_unit`. Each line of the placeholder body is enclosed in quotation marks. Quotation marks with no text inside produce blank lines at expansion.

The SET PLACEHOLDER DESCRIPTION command supplies the description `"Program unit"` for the `program_unit` placeholder.

The placeholder body for `program_unit` references other placeholders that have not yet been defined, specifically, procedure-level comments, `procedure_name`, `parameter list`, `variable_declaration`, and `statement`.

The "procedure level comments" placeholder is enclosed by optional placeholder delimiters ([]). Therefore, if you do not want to have comments at the top of your EXAMPLE program, you can erase the placeholder. LSE recognizes the fact that the placeholder is the only item in a trailing comment. After erasing the placeholder, LSE also erases the comment string and line. This way, unwanted blank comment lines are not left behind. For example:

```
PLSE NEW PLACEHOLDER "procedure level comments" NONTERMINAL EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "Procedure-level comment template" "procedure level
comments" EXAMPLE
    PLSE SET PLACEHOLDER DUPLICATION CONTEXT_DEPENDENT "procedure level comments" EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "PROCEDURE:" EXPAND 0 SPACE NEXT ADD "procedure level
comments" EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE " " EXPAND 0 SPACE NEXT ADD "procedure level comments"
EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "   {tbs}" EXPAND 0 SPACE NEXT ADD "procedure level
comments" EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE " " EXPAND 0 SPACE NEXT ADD "procedure level comments"
EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "AUTHOR:" EXPAND 0 SPACE NEXT ADD "procedure level comments"
EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE " " EXPAND 0 SPACE NEXT ADD "procedure level comments"
EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "   {tbs}" EXPAND 0 SPACE NEXT ADD "procedure level
comments" EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE " " EXPAND 0 SPACE NEXT ADD "procedure level comments"
EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "DESCRIPTION:" EXPAND 0 SPACE NEXT ADD "procedure level
comments" EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE " " EXPAND 0 SPACE NEXT ADD "procedure level comments"
EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "   {tbs}..." EXPAND 0 SPACE NEXT ADD "procedure level
comments" EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE " " EXPAND 0 SPACE NEXT ADD "procedure level comments"
EXAMPLE
```

The placeholder name can contain spaces, but if it does, the name must be put in quotation marks.

The "procedure level comments" placeholder is another nonterminal placeholder. It expands into a template for comments at the head of the procedure. Templates like this are useful for projects where you want to be sure certain information appears consistently throughout the project. In this case, there are three pieces of information: PROCEDURE, AUTHOR, and DESCRIPTION. For each category, there exists the single placeholder {tbs}. The placeholder body contains no comment characters. When you expand a multiline or vertically duplicating placeholder within a comment, the lines into which it expands have comment characters inserted automatically. For example:

```
-- [procedure level comments]
```

When you expand it, LSE recognizes that `--` is a comment string and repeats the comment string at the beginning of each line of the expansion, which produces the following:

```
-- PROCEDURE:
--
--    {tbs}
--
-- AUTHOR:
--
--    {tbs}
--
-- DESCRIPTION:
--
--    {tbs}...
--
```

The lines in the placeholder body that are a quoted-space character correspond to lines in the expansion containing only the comment delimiter. If the lines in the placeholder body were null strings, the corresponding line in the expansion would be completely empty.

Note that the `{tbs}` placeholder under `DESCRIPTION` is a list placeholder, denoted by an ellipsis (...), whereas the other two are not. List placeholders are automatically duplicated when expanded or typed over. The duplicate placeholder is always optional.

You should use required list delimiters to indicate a list that must be replaced by at least one element, for example, `{tbs}...`. You should use optional list delimiters to indicate a list that is optional, for example, `[parameter list]...`.

The `"tbs"` placeholder is a terminal placeholder. Its definition is as follows:

```
PLSE NEW PLACEHOLDER "tbs" TERMINAL EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "enter text information" "tbs" EXAMPLE
    PLSE SET PLACEHOLDER DUPLICATION VERTICAL "tbs" EXAMPLE
    PLSE SET PLACEHOLDER TERMINAL LINE "to be specified" ADD "tbs" EXAMPLE
```

Because the placeholder is a list placeholder, you should specify the DUPLICATION keyword. It tells LSE how to duplicate the placeholder. In this case, VERTICAL is specified, telling LSE to duplicate the placeholder on the next line. As with the previous example, when you type on the list `"tbs"` placeholder, it gets duplicated, and LSE automatically inserts the comment string at the beginning of the line.

When defining a terminal placeholder, the text can be as many lines as are needed to explain the correct text insertion for the placeholder. You must use quotation marks at the beginning and end of each line of text. In this case, the text `"to be specified"` is displayed when you expand the `"tbs"` placeholder. When you press any key, the text is removed from the screen.

The next placeholder to be defined is the "procedure_name" placeholder:

```
PLSE NEW PLACEHOLDER "procedure_name" TERMINAL EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "The name of a procedure" "procedure_name" EXAMPLE
    PLSE SET PLACEHOLDER AUTO SUBSTITUTE ON "procedure_name" EXAMPLE
    PLSE SET PLACEHOLDER TERMINAL LINE "A string of letters and digits starting with a
letter" ADD "procedure_name" EXAMPLE
```

The SET PLACEHOLDER AUTO SUBSTITUTE ON command specifies that during source code entry, the next occurrence of "procedure_name" is automatically replaced with the same text that you typed over the current "procedure_name" placeholder. This is useful for languages that require matching names.

In this case, the text "A string of letters and digits starting with a letter" is displayed when you expand "procedure_name".

The "parameter list" placeholder is defined next. It expands to a template corresponding to the form for a parameter list given in the syntax summary. Because it is a list placeholder, the DUPLICATION keyword is specified. The SEPARATOR keyword is also given. The definition is as follows:

```
PLSE NEW PLACEHOLDER "parameter list" NONTERMINAL EXAMPLE
    PLSE SET PLACEHOLDER AUTO SUBSTITUTE OFF "parameter list" EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "List of procedure parameters" "parameter list"
EXAMPLE
    PLSE SET PLACEHOLDER DUPLICATION CONTEXT_DEPENDENT "parameter list" EXAMPLE
    PLSE SET PLACEHOLDER PSEUDOCODE ON "parameter list" EXAMPLE
    PLSE SET PLACEHOLDER LEADING "(" "parameter list" EXAMPLE
    PLSE SET PLACEHOLDER TRAILING ")" "parameter list" EXAMPLE
    PLSE SET PLACEHOLDER SEPARATOR "; " "parameter list" EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "{param_name} : {type}" EXPAND 0 SPACE NEXT ADD "parameter
list" EXAMPLE
```

The definition of "parameter list" is described later in this section.

The DUPLICATION keyword tells LSE that the duplicated parameter list placeholder should be duplicated according to context (the context being whether the placeholder is the only item within its code fragment).

The SEPARATOR keyword indicates that a semicolon (;) is inserted in the text to separate the duplicated placeholders. Note also the trailing space on the SEPARATOR keyword. This makes a horizontal expansion more readable.

The expansion contains two placeholders, "param_name" and "type". The definition for "param_name" is as follows:

```
PLSE NEW PLACEHOLDER "param_name" TERMINAL EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "The name of a parameter" "param_name" EXAMPLE
    PLSE SET PLACEHOLDER TERMINAL LINE -
    "A string of letters and digits starting with a letter" ADD "param_name" EXAMPLE
```

The DUPLICATION keyword tells LSE that the "param_name" placeholder should be duplicated horizontally. That is, the duplicated placeholder appears on the same line, following the original placeholder.

The SEPARATOR keyword indicates that a comma (,) is inserted in the text to separate the duplicated placeholders. Note the trailing space on the SEPARATOR keyword. This makes a horizontal expansion more readable.

The TYPE placeholder is defined as a menu placeholder, as follows:

```
PLSE NEW PLACEHOLDER TYPE MENU EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "Data Type" TYPE EXAMPLE
    PLSE SET PLACEHOLDER DUPLICATION CONTEXT_DEPENDENT TYPE EXAMPLE
    PLSE SET PLACEHOLDER MENU LINE "INTEGER" "Integer data type" TEXT OFF NEXT ADD TYPE EXAMPLE
    PLSE SET PLACEHOLDER MENU LINE "BOOLEAN" "Boolean data type" TEXT OFF NEXT ADD TYPE EXAMPLE
```

When TYPE is expanded, a menu is displayed in the buffer. The placeholder body for a menu placeholder is defined in a similar way as a nonterminal placeholder and contains the elements of the menu.

The expansion of "variable_declaration" looks much like the expansion of parameter list. The difference is the addition of an optional initial_value placeholder. In the EXAMPLE language, the placeholder enables you to optionally initialize the variables you are declaring, as follows:

```
PLSE NEW PLACEHOLDER "variable_declaration" NONTERMINAL EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "Declares a variable" "variable_declaration" EXAMPLE
    PLSE SET PLACEHOLDER DUPLICATION VERTICAL "variable_declaration" EXAMPLE
    PLSE SET PLACEHOLDER TRAILING ";" "variable_declaration" EXAMPLE
    PLSE SET PLACEHOLDER SEPARATOR "; " "variable_declaration" EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "{identifier}... : {type} := [initial_value]..." -
            EXPAND 0 SPACE NEXT ADD "variable_declaration" EXAMPLE
```

If you do not have any variables to declare in your program, you can erase the optional "variable_declaration" placeholder, but you do not want to leave the semicolon (;) behind. The TRAILING keyword (in the definition of variable_declaration TRAILING ";") specifies that the string ";" be associated with the "variable_declaration" placeholder. If the placeholder is deleted, LSE looks immediately to the right (not including blank space). If it sees the trailing string, LSE deletes it.

The "initial_value" placeholder is defined as follows:

```
PLSE NEW PLACEHOLDER "initial_value" TERMINAL EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "Variable declaration initial value" -
        "initial_value" EXAMPLE
    PLSE SET PLACEHOLDER DUPLICATION HORIZONTAL "initial_value" EXAMPLE
    PLSE SET PLACEHOLDER LEADING ":=" "initial_value" EXAMPLE
    PLSE SET PLACEHOLDER SEPARATOR ", " "initial_value" EXAMPLE
    PLSE SET PLACEHOLDER TERMINAL LINE -
        "The initial value(s) of identifier(s) - either Integer or Boolean value(s)" -
        NEXT ADD "initial_value" EXAMPLE
```

If you do not want to initialize the variables, you can erase the optional placeholder, but you do not want to leave the ":=" behind. This is similar to the ";" after the "variable_declaration" placeholder. In this case, LEADING ":=" specifies that the string ":=" be associated with the "initial_value" placeholder. If the placeholder is deleted, LSE looks immediately to the left (not including blank space). If it sees the leading string, LSE deletes it. A placeholder can have both leading and trailing text strings associated with it.

The preceding descriptions of the LEADING and TRAILING keywords are not quite complete. Based on the descriptions given thus far, the definition of "parameter_list", repeated here for convenience, would appear to be wrong:

```
PLSE NEW PLACEHOLDER "parameter list" NONTERMINAL EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "List of procedure parameters" "parameter list"
EXAMPLE
    PLSE SET PLACEHOLDER DUPLICATION CONTEXT_DEPENDENT "parameter list" EXAMPLE
    PLSE SET PLACEHOLDER LEADING  "("   "parameter list" EXAMPLE
    PLSE SET PLACEHOLDER TRAILING  ")"  "parameter list" EXAMPLE
    PLSE SET PLACEHOLDER SEPARATOR "; " "parameter list" EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "{param_name} : {type}" -
        EXPAND 0 SPACE NEXT ADD "parameter list" EXAMPLE
```

For example, after a few expansions of program_unit, the following code fragment might result:

```
PROCEDURE x (a : INTEGER; [parameter list]...) IS
```

Deleting optional placeholders based on the previous description incorrectly leaves the following:

```
PROCEDURE X (a : INTEGER IS
```

That is, both the separator string ";", and the trailing string "") have been removed. However, if you go through these steps using the EXAMPLE language provided with LSE, you find that this does not happen. What actually results is correct, as follows:

```
PROCEDURE X (a : INTEGER) IS
```

The full procedure that LSE goes through when a placeholder is deleted is as follows:

1. Is the placeholder in a comment? If so, delete the comment characters as well (leading, trailing, and separator characters are not used in this situation).
2. Is a separator defined for the placeholder? Is the placeholder preceded by the separator? If so, delete it and skip the next step.
3. Have leading or trailing strings been defined for the placeholder? If so, look for the leading and trailing strings, and if they are found, delete them.
4. If, after these deletions, nothing or only punctuation characters remain on the line, delete the blank space except one blank.
5. Finally, if one or both characters immediately to the left and right of the deletion are punctuation characters, delete all blank space.

In the example, Step 2 causes the `"parameter_list"` placeholder to work as you want it to. Because the placeholder was expanded, a separator character was present. After LSE found and deleted it, LSE did not go on to look for leading or trailing strings. Thus, the parentheses correctly remained around the parameter.

The `"statement"` placeholder is defined as a menu placeholder. It is also used as a list placeholder, as specified in the `program_unit` placeholder definition:

```
PLSE NEW PLACEHOLDER "statement" MENU EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "EXAMPLE statements" "statement" EXAMPLE
    PLSE SET PLACEHOLDER DUPLICATION VERTICAL "statement" EXAMPLE
    PLSE SET PLACEHOLDER TRAILING ";" "statement" EXAMPLE
    PLSE SET PLACEHOLDER SEPARATOR "; " "statement" EXAMPLE
    PLSE SET PLACEHOLDER MENU LINE "ASSIGNMENT" "" TOKEN OFF NEXT ADD "statement" EXAMPLE
    PLSE SET PLACEHOLDER MENU LINE "IF" "" TOKEN OFF NEXT ADD "statement" EXAMPLE
    PLSE SET PLACEHOLDER MENU LINE "LOOP" "" TOKEN OFF NEXT ADD "statement" EXAMPLE
    PLSE SET PLACEHOLDER MENU LINE "EXIT" "" TOKEN OFF NEXT ADD "statement" EXAMPLE
```

The DUPLICATION keyword tells LSE that the `"statement"` list placeholder should be duplicated vertically. The SEPARATOR keyword indicates that a semicolon is used to separate the duplicated placeholders.

The SET PLACEHOLDER MENU LINE commands indicate that ASSIGNMENT, IF, LOOP, and EXIT are tokens. They are defined as tokens so you can type ASSIGNMENT, IF, LOOP, or EXIT directly into the buffer.

When "statement" is expanded, a menu is displayed and another copy of statement with optional list delimiters is inserted on the next line in the buffer. When this menu is displayed, the text strings associated with the DESCRIPTION keyword on the token definitions is displayed as well. These text strings are also displayed when you enter the SHOW TOKEN command.

Token Definitions for the EXAMPLE Language

Tokens are defined for keywords or punctuation characters that you want to type directly into the buffer. When expanded, they provide templates for corresponding language constructs.

ASSIGNMENT, IF, LOOP, and EXIT are defined as tokens. To define a token, use the NEW TOKEN command, as follows:

```
PLSE NEW TOKEN ASSIGNMENT TERMINAL EXAMPLE
    PLSE SET TOKEN DESCRIPTION "Assignment statement" ASSIGNMENT EXAMPLE
    PLSE SET TOKEN BODY LINE "{identifier} := {expression}" -
        CURRENT 0 SPACE NEXT ADD ASSIGNMENT EXAMPLE
```

The DESCRIPTION keyword supplies the description "Assignment statement" for the token ASSIGNMENT. This text string appears in the menu when you expand {statement} . . . as well as when you enter the SHOW TOKEN command.

Token names can consist of any combination of characters. However, token names cannot have leading or trailing blank space. This feature enables you to define tokens with names that are suited for the context in which they will be used. It would be useful to define the assignment operator (:=) as a token so that if you want to insert an assignment operation into an editing buffer, you can type the assignment operator (:=) followed by the EXPAND key and have {identifier} := {expression} placed in the editing buffer.

In addition to the ASSIGNMENT token, the assignment operator (:=) is defined to be a token, as follows:

```
PLSE NEW TOKEN ":=" TERMINAL EXAMPLE
    PLSE SET TOKEN DESCRIPTION "Assignment statement" ":=" EXAMPLE
    PLSE SET TOKEN BODY LINE "{identifier} := {expression}" CURRENT 0 SPACE NEXT ADD ":=" -
        EXAMPLE
```

In the following example, the IF statement contains an optional ELSE statement_list construct, as indicated in the syntax summary. Therefore, you must include the construct in the IF token definition and define an additional placeholder for ELSE statement_list, as follows:

```
PLSE NEW TOKEN IF TERMINAL EXAMPLE
    PLSE SET TOKEN DESCRIPTION "IF {expression} THEN..." IF EXAMPLE
    PLSE SET TOKEN BODY LINE "IF {boolean_exp} " CURRENT 0 SPACE NEXT ADD IF EXAMPLE
    PLSE SET TOKEN BODY LINE "THEN" CURRENT 0 SPACE NEXT ADD IF EXAMPLE
    PLSE SET TOKEN BODY LINE " {statement}...;" CURRENT 0 SPACE NEXT ADD IF EXAMPLE
    PLSE SET TOKEN BODY LINE "[ELSE {statement}...]" CURRENT 0 SPACE NEXT ADD IF EXAMPLE
    PLSE SET TOKEN BODY LINE "END IF" CURRENT 0 SPACE NEXT ADD IF EXAMPLE
```

The definition for the LOOP token is as follows:

```
PLSE NEW TOKEN LOOP TERMINAL EXAMPLE
    PLSE SET TOKEN DESCRIPTION "[loop_id]: LOOP ... END LOOP;" LOOP EXAMPLE
    PLSE SET TOKEN BODY LINE "[loop_id]: LOOP" CURRENT 0 SPACE NEXT ADD LOOP EXAMPLE
    PLSE SET TOKEN BODY LINE " {statement}...;" CURRENT 0 SPACE NEXT ADD LOOP EXAMPLE
    PLSE SET TOKEN BODY LINE "END LOOP" CURRENT 0 SPACE NEXT ADD LOOP EXAMPLE
```

The colon (:) following the optional [loop_id] placeholder presents a problem similar to the ";" after the "variable_declaration" placeholder. That is, if the optional placeholder is erased the colon is left behind. For example:

```
PLSE NEW PLACEHOLDER "loop_id" TERMINAL EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "LOOP LABEL" "loop_id" EXAMPLE
    PLSE SET PLACEHOLDER TRAILING ":" "loop_id" EXAMPLE
    PLSE SET PLACEHOLDER TERMINAL LINE "A string of letters and digits starting with a letter." -
        NEXT ADD "loop_id" EXAMPLE
```

Once again, use the TRAILING keyword. If the "loop_id" placeholder is erased, LSE looks immediately to the right for the colon (:). If it is found, it is erased.

Finally, the definition for the token EXIT follows:

```
PLSE NEW TOKEN EXIT TERMINAL EXAMPLE
    PLSE SET TOKEN DESCRIPTION "EXIT [loop_id] WHEN ... " EXIT EXAMPLE
    PLSE SET TOKEN BODY LINE "EXIT [loop_id] WHEN {boolean_exp} " -
        CURRENT 0 SPACE NEXT ADD EXIT EXAMPLE
```

All placeholders referenced in token definitions must be defined. The "identifier" placeholder is a terminal placeholder and is defined as follows:

```
PLSE NEW PLACEHOLDER IDENTIFIER TERMINAL EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "Identifier" IDENTIFIER EXAMPLE
    PLSE SET PLACEHOLDER DUPLICATION HORIZONTAL IDENTIFIER EXAMPLE
    PLSE SET PLACEHOLDER SEPARATOR ", " IDENTIFIER EXAMPLE
    PLSE SET PLACEHOLDER TERMINAL LINE "A string of letters and digits starting with a letter." -
        NEXT ADD IDENTIFIER EXAMPLE
```

In the syntax summary, EXPRESSION is specified as a menu placeholder and is defined as follows:

```
PLSE NEW PLACEHOLDER EXPRESSION MENU EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "Boolean or arithmetic expression" EXPRESSION EXAMPLE
    PLSE SET PLACEHOLDER MENU LINE "boolean_exp" "use AND OR NOT" PLACEHOLDER -
        OFF NEXT ADD EXPRESSION EXAMPLE
    PLSE SET PLACEHOLDER MENU LINE "arithmetic_exp" "use + - * /" PLACEHOLDER -
 OFF NEXT ADD EXPRESSION EXAMPLE
```

Use the DESCRIPTION keyword to specify that "boolean_exp" and "arithmetic_exp" are placeholders. As such, they need to be defined. When you expand {expression} and select one of the menu options, the DESCRIPTION keyword causes the expansion of the menu item to be placed into the buffer, not the menu item itself. For example:

```
PLSE NEW PLACEHOLDER "boolean_exp" NONTERMINAL EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "Boolean expression" "boolean_exp" EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "[NOT] {identifier} [relop identifier]" -
        EXPAND 0 SPACE NEXT ADD "boolean_exp" EXAMPLE

PLSE NEW PLACEHOLDER "arithmetic_exp" TERMINAL EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "arithmetic expression" "arithmetic_exp" EXAMPLE
    PLSE SET PLACEHOLDER TERMINAL LINE "An arithmetic expression" NEXT ADD "arithmetic_exp" -
        EXAMPLE
```

For example, if you select boolean_exp from the menu, the following is inserted into the buffer:

```
[NOT] {identifier} [relop identifier]
```

If the TEXT keyword is used with MENU LINE, the literal menu item is inserted. If this is formatted as a placeholder, {boolean_exp}, the string {boolean_exp} is placed in the buffer.

You would have to expand {boolean_exp} again to have it inserted into the buffer. Because "arithmetic_exp" is a terminal placeholder, selecting it causes {arithmetic_exp} to appear in the buffer along with its expanded description. The same thing occurred earlier in the definition of statement. For example, ASSIGNMENT was a menu item. Choosing ASSIGNMENT from the menu causes the expansion of the ASSIGNMENT token to appear in the buffer.

If you do not want automatic expansion, the definition of EXPRESSION appears as follows:

```
PLSE NEW PLACEHOLDER EXPRESSION MENU EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "Boolean or arithmetic expression" EXPRESSION EXAMPLE
    PLSE SET PLACEHOLDER MENU LINE "{boolean_exp}" "use AND OR NOT" TEXT -
        OFF NEXT ADD EXPRESSION EXAMPLE
    PLSE SET PLACEHOLDER MENU LINE "{arithmetic_exp}" "use + - * /" TEXT -
        OFF NEXT ADD EXPRESSION EXAMPLE
```

NOT is a keyword, therefore it should be defined as a token. However, it is also used as a placeholder in the expansion of {boolean_exp}. This is an example of a situation in which you could use the alternate form of the NEW TOKEN command, as follows:

```
PLSE NEW TOKEN NOT ALIAS EXAMPLE
    PLSE SET TOKEN PLACEHOLDER NOT EXAMPLE

PLSE NEW PLACEHOLDER NOT NONTERMINAL EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "Keyword NOT" NOT EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "NOT" EXPAND 0 SPACE NEXT ADD NOT EXAMPLE
```

The token NOT takes its parameters from the placeholder NOT. That is, even though it is not explicitly specified, the token is associated with the EXAMPLE language and has the description "Keyword NOT".

The final placeholders are defined as follows:

```
PLSE NEW PLACEHOLDER "relop identifier" NONTERMINAL EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "Optional part of a boolean expression" -
        "relop identifier" EXAMPLE
    PLSE SET PLACEHOLDER BODY LINE "{relop} {identifier}" -
        EXPAND 0 SPACE NEXT ADD "relop identifier" EXAMPLE

PLSE NEW PLACEHOLDER "relop" MENU EXAMPLE
    PLSE SET PLACEHOLDER DESCRIPTION "relative operator" "relop" EXAMPLE
    PLSE SET PLACEHOLDER MENU LINE "=" ""  TEXT OFF NEXT ADD "relop" EXAMPLE
    PLSE SET PLACEHOLDER MENU LINE "<" ""  TEXT OFF NEXT ADD "relop" EXAMPLE
    PLSE SET PLACEHOLDER MENU LINE ">" ""  TEXT OFF NEXT ADD "relop" EXAMPLE
    PLSE SET PLACEHOLDER MENU LINE "<=" "" TEXT OFF NEXT ADD "relop" EXAMPLE
    PLSE SET PLACEHOLDER MENU LINE ">=" "" TEXT OFF NEXT ADD "relop" EXAMPLE
```

Additional placeholders for the arithmetic expressions are not defined. Rather, the `"arithmetic_exp"` placeholder is defined as a terminal placeholder to notify you that an arithmetic expression is required. A choice such as this is strictly up to the author of the templates. It all depends on the level of detail you want. A relevant criterion might be, for example, the knowledge of the people using the templates.

Now that the template definition is complete, you must execute the commands in your source file. See Section 4.4 for details on executing your source file and creating an environment file for later use.

4.2.3 Redefining Language Elements

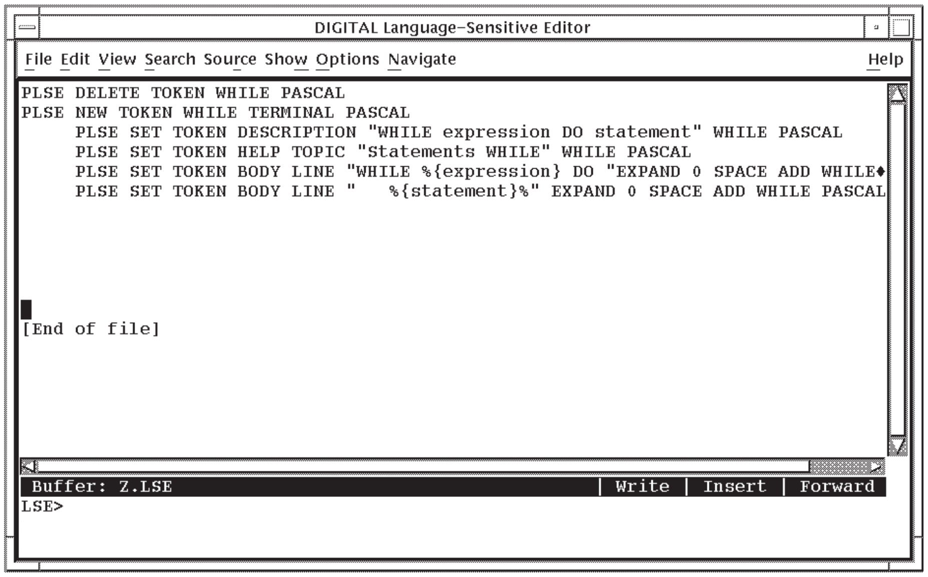
You can redefine token, placeholder, package, and language definitions interactively. Thus, you can add or delete constructs, reformat menus, or edit descriptions within existing definitions. The EXTRACT command places the current definition of a token, placeholder, package, or language in the current buffer. You can then make the appropriate modifications and execute the new definitions.

To redefine a token, placeholder, package, or language, do the following:

1. Enter the NEW BUFFER command and a new buffer name to enter an empty buffer.
2. Enter the EXTRACT TOKEN, EXTRACT PLACEHOLDER, EXTRACT PACKAGE, or EXTRACT LANGUAGE command, followed by the name of the token, placeholder, or language.
3. Edit the definition of the selected token, placeholder, package, or language.
4. Enter the EXECUTE BUFFER LSE command to execute the new definition.

When redefining tokens or placeholders, the previous definitions must be deleted. As shown in Figure 4–2, the EXTRACT command automatically puts a DELETE command before the NEW command when it places the definition in the buffer. The DELETE TOKEN command deletes the previous definition of a token, and the NEW TOKEN command provides a new definition.

Figure 4–2 Extracting a Token


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Navigate
Help

```
PLSE DELETE TOKEN WHILE PASCAL
PLSE NEW TOKEN WHILE TERMINAL PASCAL
     PLSE SET TOKEN DESCRIPTION "WHILE expression DO statement" WHILE PASCAL
     PLSE SET TOKEN HELP TOPIC "Statements WHILE" WHILE PASCAL
     PLSE SET TOKEN BODY LINE "WHILE %{expression} DO "EXPAND 0 SPACE ADD WHILE♦
     PLSE SET TOKEN BODY LINE "   %{statement}%" EXPAND 0 SPACE ADD WHILE PASCAL




[End of file]

 Buffer: Z.LSE                                         | Write | Insert | Forward
LSE>
```

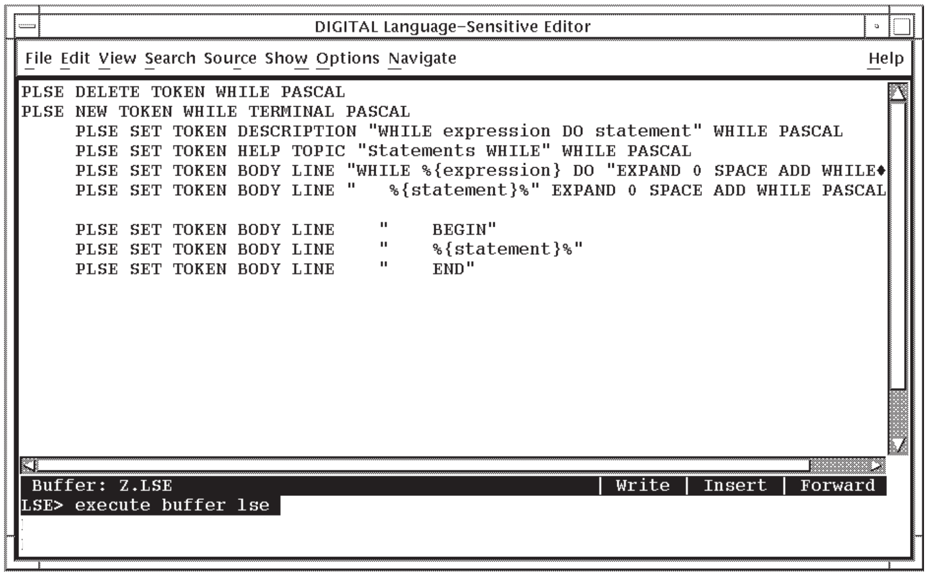
For example, if you want to add a BEGIN/END construct to the definition of a Pascal WHILE statement, do the following:

1. Enter the following command while in an empty buffer:

   ```
   EXTRACT TOKEN WHILE PASCAL
   ```

2. Press Ctrl/Z to remove the command prompt and edit the token by adding BEGIN and END statements.
3. Press Ctrl/Z to get the command prompt after you have the definition the way you want it.
4. Enter the EXECUTE BUFFER LSE command to execute the new definition, as shown in Figure 4–3.

Figure 4–3 Executing a New Definition


DIGITAL Language–Sensitive Editor
File Edit View Search Source Show Options Navigate Help

```
PLSE DELETE TOKEN WHILE PASCAL
PLSE NEW TOKEN WHILE TERMINAL PASCAL
     PLSE SET TOKEN DESCRIPTION "WHILE expression DO statement" WHILE PASCAL
     PLSE SET TOKEN HELP TOPIC "Statements WHILE" WHILE PASCAL
     PLSE SET TOKEN BODY LINE "WHILE %{expression} DO "EXPAND 0 SPACE ADD WHILE♦
     PLSE SET TOKEN BODY LINE "   %{statement}%" EXPAND 0 SPACE ADD WHILE PASCAL

     PLSE SET TOKEN BODY LINE    "    BEGIN"
     PLSE SET TOKEN BODY LINE    "    %{statement}%"
     PLSE SET TOKEN BODY LINE    "    END"
```

```
 Buffer: Z.LSE                                    | Write | Insert | Forward
LSE> execute buffer lse
```

Thereafter, each time you use the WHILE token in Pascal in the current editing session, LSE provides you with the new definition.

You can use the SET LANGUAGE commands as an alternate method for redefining language definitions. This method works best when you want to modify just one or two keywords in your language definition, or the attributes of all the languages.

For example, if you want to change the setting of the EXPAND_CASE keyword from AS_IS to UPPER, enter the following command:

```
SET LANGUAGE EXPAND_CASE UPPER PASCAL
```

The SET LANGUAGE command lets you modify your language definition without having to execute the language definition again.

4.3 Defining Adjusted Indentation

Adjusted indentation is a defined value you give to assign relative position to program constructs so they can be outlined. You cannot see adjusted indentation because LSE does not move any text; LSE treats lines as though the indentation has changed.

Before you define adjustment values for overviews, see the default adjustment values for the programming language you want to use by entering the SHOW ADJUSTMENT * command.

When you define adjustment values to match your own coding conventions, the following criteria might be helpful for judging the quality of the overviews you can generate with your adjustments:

- Each major language construct should be compressible to a single line.
- Comments should conceal the constructs they describe.
- Lower-level constructs should never hide higher-level constructs.

Adjustments are defined on the LSE command line. Press F16 (the Do key) to make the command-line prompt appear. See the *HP DECset for OpenVMS Language-Sensitive Editor/ Source Code Analyzer Reference Manual* for information about the SET ADJUSTMENT command and its qualifiers, or type HELP LSE SET ADJUSTMENT at the system prompt for online information.

The following sections describe several methods for adjusting indentation for outlining your programs.

4.3.1 Adjusting Single Lines

The SET ADJUSTMENT CURRENT *n* command sets the current line to the indentation value you specify. This is a logical value; the actual position of the text on the line does not change. For example, the following statement shows the IF, THEN, and ELSE clauses in the same column:

```
IF P
THEN
    ADD 5 TO K
    END-ADD
ELSE
    ADD 6 TO K
    END-ADD
```

By entering the following commands, LSE treats the actual text as if it looked like the subsequent text:

```
SET ADJUSTMENT CURRENT 1 then
SET ADJUSTMENT CURRENT 1 else
```

```
IF P
 THEN
    ADD 5 TO K
    END-ADD
 ELSE
    ADD 6 TO K
    END-ADD
```

With the adjusted indentation, the actual code can be compressed into the following overview line:

```
«IF P»
```

You can adjust the indentation of a line to the left instead of to the right by supplying a negative value to the SET ADJUSTMENT CURRENT command. For example:

```
SET ADJUSTMENT CURRENT -1 "*"
```

Comments should frequently be adjusted to the left so programming code can be concealed by an overview line that contains the comment about the code.

4.3.2 Adjusting Multiple Lines

If a program construct does not have an indented body, you can specify a pair of SET ADJUSTMENT SUBSEQUENT *n* commands to logically indent the whole construct. For example, if the lines between the BLISS BEGIN and END statement are not indented, you can adjust for that with the following definitions:

```
SET ADJUSTMENT SUBSEQUENT  1 begin
SET ADJUSTMENT SUBSEQUENT -1 end
```

Specifying a 1 for BEGIN logically indents all subsequent lines by one; specifying a −1 resets lines after the END clause to their original position.

Adjustments are cumulative. If nested BEGIN-END statements occurred, lines after each BEGIN clause would be indented by one and moved to the left after each END clause.

When you use the SET ADJUSTMENT SUBSEQUENT command, be sure the total of the values for a single construct add up to zero. For example, in Ada and Pascal, the PROCEDURE keyword sometimes has a matching END and sometimes does not. Therefore, the SET ADJUSTMENT SUBSEQUENT command should not be applied to PROCEDURE.

You do not need to specify the SET ADJUSTMENT SUBSEQUENT command in pairs, with values of N and −N. Many definitions are acceptable as long as the combined values add up to zero.

You should choose whether to adjust language clauses and constructs to the left or right depending on how you want the interaction of an adjusted indentation definition with other definitions, particularly those involving comments. Moving comments to the left and moving code to the right works well for most situations.

4.3.3 Adjusting Lines in Programming Languages Without Indentation

Languages without indentation can still take advantage of overviews. For example, products such as HP DECdocument use files containing markup-language tags. Many of these tags are constructs with well-marked beginnings and ends that can be treated as follows:

```
SET ADJUSTMENT SUBSEQUENT 2 "<NOTE>"
SET ADJUSTMENT SUBSEQUENT  -2 "<ENDNOTE>"
SET ADJUSTMENT CURRENT  -1 "<ENDNOTE>"
```

You can treat HP DECdocument markup-language constructs that do not have visible endings as follows:

```
SET ADJUSTMENT CURRENT  -1 "<CHAPTER>"
SET ADJUSTMENT CURRENT  -1 "<HEAD1>"
SET ADJUSTMENT CURRENT  -1 "<HEAD2>"
SET ADJUSTMENT CURRENT  -1 "<HEAD3>"
```

4.3.4 Preventing Text Compression into Overview Lines

The SET ADJUSTMENT COMPRESS OFF command does not allow a specified construct to be compressed into an overview line. For example, you might decide that LSE should not compress BEGIN constructs into overview lines like the following:

```
«BEGIN»
```

The following definition instructs LSE to display the detailed text instead of the overview line:

```
SET ADJUSTMENT COMPRESS OFF begin
```

Note

Programming-language constructs can be concealed in overview lines that are compressed from higher-level constructs.

4.3.5 Defining Appropriate Overview Text

LSE typically uses the first line of a group of text to form an overview line. Because this text might not be appropriate (for example, if it is a blank line, an empty comment, or a row of asterisks), you can explicitly control what text appears in overview lines by using the SET ADJUSTMENT OVERVIEW command. If a string is inappropriate as overview text, you can define it using the SET ADJUSTMENT OVERVIEW OFF command. For example, suppose a HP C program is formatted as follows:

```
char *
getline(row)
int row;
{
    ...
}
```

If each C function is compressed to a single line, `char *` becomes the overview text despite its uninformative nature. The text `getline` is the function name and should be the overview text. To prevent `char *` from appearing as the overview text, use the following definition:

```
SET ADJUSTMENT OVERVIEW OFF "char *"
```

4.3.6 Inheriting Indentation Values

In some situations, the visible indentation of a line is not a good indication of what its adjusted indentation should be. The visible indentation of a blank line is always 0. The visible indentation of a FORTRAN or COBOL comment line is usually 1. In these situations, you can use the SET ADJUSTMENT INHERIT command to cause a line to inherit its visible indentation from the adjusted indentation of a neighboring line.

The SET ADJUSTMENT INHERIT command is typically used in adjustments for blank lines, form feeds, conditional compilations, and labeled lines. It has the following keywords:

Keyword	Description
PREVIOUS	The visible indentation for the current line is taken from the adjusted indentation of the previous line.
NEXT	The visible indentation for the current line is taken from the adjusted indentation of the next line.
MINIMUM	The visible indentation for the current line is taken from the adjusted indentation of either the previous line or the next line, whichever is smaller.
MAXIMUM	The visible indentation for the current line is taken from the adjusted indentation of either the previous line or the next line, whichever is larger.

With C conditional compilation, a common coding convention is to type the conditional compilation lines at the left margin, although the corresponding program code is indented. For example:

```
SET ADJUSTMENT INHERIT NEXT "$(COLUMN=1)#if"
```

When you use this definition, LSE takes the indentation of a conditional compilation line from the indentation of the line below it.

4.3.7 Grouping Comment Lines

You can treat contiguous comment lines as a single unit if they have the same adjusted indentation by specifying the SET ADJUSTMENT UNIT ON command. For example:

```
    ! Store away the info needed to update
    ! the history.
    Update_info::Edit_no = Edit_no
    Update_info::Version_record = Rec_num
```

Without grouping the comments into a unit, the second comment in this example becomes the overview line for the assignments. To treat both comment lines as one unit, and to have the first comment become the overview for both comment lines and the associated statements, combine the comment lines into one group by using the following commands:

```
SET ADJUSTMENT UNIT ON
SET ADJUSTMENT CURRENT= -1 "!"
```

4.3.8 Using Prefixes in Adjustments

In some situations, such as a label construct, you want to avoid having the starting text of a line determine indentation or influence adjustment. Use the SET ADJUSTMENT PREFIX *"prefix-text"* command to ignore specified text at the beginning of lines.

The following example instructs LSE to treat an identifier followed by a colon as a prefix, if a BEGIN construct follows the identifier and colon on a source line:

```
SET ADJUSTMENT PREFIX ADJUSTMENT FOLLOWING "$(IDENTIFIER):$(PREFIX)BEGIN"
```

The BEGIN clause is the first text after the prefix.

4.3.9 Adjusting Blank Lines

The default adjustment for blank lines is as follows:

```
SET ADJUSTMENT INHERIT MAXIMUM
SET ADJUSTMENT UNIT ON
SET ADJUSTMENT COUNT OFF
```

This usually causes a blank line to be absorbed into a neighboring group of source code lines. For example:

```
BEGIN

  a = b;
```

In this example, the blank line following BEGIN inherits the adjusted indentation of the `a = b` line because that line is indented more than the BEGIN line. Therefore, the blank line is part of the group that starts with the BEGIN line.

Consider the following code fragment:

```
IF something
 THEN
    a = b;

 ELSE
    c = d;
```

The blank line preceding `ELSE` inherits the adjusted indentation of the `a = b` line. Therefore, the blank line is part of the group that starts with the `THEN` line.

To redefine the behavior of blank lines, you must define an adjustment for the `LINE_END` predefined pattern. For example:

```
NEW ADJUSTMENT "$(LINE_END)"
SET ADJUSTMENT UNIT ON
SET ADJUSTMENT COUNT OFF
SET ADJUSTMENT INHERIT NEXT
SET ADJUSTMENT CURRENT -1
SET ADJUSTMENT COMPRESS OFF
```

You might use this adjustment if you want a blank line to terminate a group started by a preceding comment line. For example, suppose you use this definition with the following code fragment:

```
! comment text
 IF something
  THEN
    a = b;

 c = d;
```

The group started by the comment line will include only the lines through the `a = b` line.

In languages where there is little syntax to distinguish program constructs, such as HP C, you can define empty space to be significant. For example:

```
NEW ADJUSTMENT "$(LINE_END)"
SET ADJUSTMENT CURRENT -1
NEW ADJUSTMENT "$(FORMFEED)"
SET ADJUSTMENT CURRENT -2
```

This definition causes every blank line to have an adjusted indentation of -1 because its visible indentation is always 0, and every form-feed line to have an adjusted indentation of -1 (if each form feed is in column 1).

4.3.10 Adjusting Bracketed Comments

Language compilers and LSE do not necessarily have the same definitions for comments. LSE treats a line as a comment only if it includes comment delimiting text. The second line in the following example is not considered a comment by LSE:

```
/*
   Open the file.
*/
```

Consider the following adjustment definition:

```
NEW ADJUSTMENT "/*"
SET ADJUSTMENT CURRENT -1
```

This particular style does not present a problem for the overview facility. Because the inner lines of a comment are indented further than the comment delimiter text, LSE treats the whole comment as one group.

You can write definitions for `/*` and `*/` with the /SUBSEQUENT qualifier to cause intervening lines to indent. Use this approach with caution, however, because if `*/` does not appear at the beginning of a line, LSE cannot detect the `*/`.

4.3.11 Adjusting Fixed Comments

In some languages, such as FORTRAN and COBOL, the indentation of a comment is of no interest because it is in a fixed column. One way to handle this situation is to adjust the comment to inherit the indentation from the line after the comment line. To do this, use the INHERIT keyword, as follows:

```
NEW ADJUSTMENT "$(COLUMN=1)C"
SET ADJUSTMENT INHERIT NEXT
SET ADJUSTMENT CURRENT -1
SET ADJUSTMENT UNIT ON
```

Another approach is to ignore the comment marker and take the visible indentation from the text that follows on the line. To do this, use the PREFIX keyword, as follows:

```
NEW ADJUSTMENT "$(COLUMN=1)*$(PREFIX)"
SET ADJUSTMENT PREFIX FOLLOWING CURRENT
```

See Section 3.5.2 for FORTRAN-specific information.

4.4 Saving Language Definitions

To create and save an environment file for a new language, do the following:

1. Create your source file in an empty buffer. The source file should have an extension of .LSE.
2. Put all the necessary LSE language, token, and placeholder definitions into that source file.
3. Execute the commands in the source file with the EXECUTE BUFFER LSE command while in LSE. This loads the definitions into the current editing session, but does not save them.
4. Enter the SAVE ENVIRONMENT command while in LSE. This command saves your definitions and produces a binary file with a file extension of .ENV.

The SAVE ENVIRONMENT command writes out all user-defined languages, placeholders, tokens, and aliases to an editing environment file. This command executes immediately to save the environment active at the moment you enter the command.

For information on OpenVMS systems, refer to the *HP DECset for OpenVMS Language-Sensitive Editor / Source Code Analyzer Reference Manual*.

4.5 Using Aliases

An **alias** is an abbreviation for a long text string or identifier that you want to expand repeatedly into your source code. When you expand an alias, it is replaced with the text string you defined for the alias.

For example, you can define the alias `PCN` by pressing F16 (the Do Key) and specifying the following command:

```
NEW ALIAS PCN PAS_COPY_NNT
```

After you define the alias, you can type the alias and expand it with Ctrl/E.

Aliases that contain nonalphanumeric characters must be defined with quotation marks. For example:

```
NEW ALIAS INCR "X = X + 1"
```

You can define an alias by positioning the text cursor on an identifier or text string and pressing PF1-Ctrl/A. LSE prompts you for the alias name to be assigned to your selection.

4.6 Using LSE Packages

LSE provides templates for subroutine packages. These packages define OpenVMS System Services, Run-Time Library (LIB$, STR$, SMG$), and OpenVMS Record Management System (HP RMS) routines. In addition, LSE provides a mechanism for defining packages for your own subroutine libraries.

You can use the SCA REPORT command to generate package definitions automatically for your subroutine libraries. See the *Guide to Detailed Program Design for OpenVMS Systems* for more information.

You can use the following commands to create package definitions:

NEW PACKAGE
NEW ROUTINE

The NEW PACKAGE command defines a subroutine package for which subroutine-call templates are automatically generated. Packages contain routine definitions that describe calls to subroutines, and the parameters for the calls. The NEW ROUTINE command defines templates for a routine within a subroutine package. This command makes the routine an element of a package.

The following is an example of a package definition:

```
PLSE NEW PACKAGE SYSTEM_SERVICES
PLSE SET PACKAGE LANGUAGE BASIC   ADD SYSTEM_SERVICES
PLSE SET PACKAGE LANGUAGE C       ADD SYSTEM_SERVICES
PLSE SET PACKAGE LANGUAGE C++     ADD SYSTEM_SERVICES
PLSE SET PACKAGE LANGUAGE COBOL   ADD SYSTEM_SERVICES
PLSE SET PACKAGE LANGUAGE DCL     ADD SYSTEM_SERVICES
PLSE SET PACKAGE LANGUAGE FORTRAN ADD SYSTEM_SERVICES
PLSE SET PACKAGE LANGUAGE PLI     ADD SYSTEM_SERVICES
PLSE SET PACKAGE HELP LIBRARY helplib SYSTEM_SERVICES
PLSE SET PACKAGE HELP TOPIC "System_Services  "         SYSTEM_SERVICES
PLSE SET PACKAGE ROUTINE EXPAND "LSE$PKG_EXPAND_ROUT_"   SYSTEM_SERVICES !Special routines
PLSE SET PACKAGE PARAMETER EXPAND "LSE$PKG_EXPAND_PARM_" SYSTEM_SERVICES !for system services
```

LSE provides two sets of predefined HP DECTPU routines to help you write your own packages. However, you might want to create your own HP DECTPU procedures to help you write more complex packages.

The OpenVMS System Services and HP RMS packages consist of routine definitions and parameter definitions that are available automatically when you use LSE with any of the supported programming languages. See the Software Product Description (SPD) for a complete list of programming languages that LSE supports.

Routines are language-independent and are useful for describing subroutine libraries. They need to be defined only once. Routine names are used in the same way tokens are used. For example, if you type the routine name `SYS$FILESCAN` and expand it, you get the following results:

```
sys$filescan    ( {srcstr},
                  {valuelst},
                  [fldflags] )
```

Most languages access OpenVMS System Services and HP RMS routines with the prefix SYS$. These languages must use the SYSTEM_SERVICES package. Other languages use different prefixes. For example, Ada prohibits the dollar sign prefix ($) and must use the STARLET package. BLISS and Pascal require the dollar sign prefix ($) and must use the KEYWORD_SYSTEM_SERVICES package.

For example, to call the $SNDOPR system service from a PL/I program, type `status := sys$sndopr` and press Ctrl/E with the text cursor positioned after `sys$sndopr` to expand it to the following text:

```
status := sys$sndopr (
              {msgbuf},
              [chan])
```

This indicates that the `$SNDOPR` system service has two parameters: `msgbuf` (required) and `chan` (optional). Because `chan` is optional, LSE expands it with an optional placeholder that you can either delete or expand. Languages other than Ada and BLISS have similar features.

In Ada, the dollar sign is not used as part of the system service name. For example, you can type `starlet.sndopr,` then press Ctrl/E to expand it to the following text:

```
STARLET.SNDOPR (
    STATUS => {status},
    MSGBUF => {msgbuf},
    [CHAN   => {chan}]);
```

In BLISS, the system services start with a dollar sign without the leading SYS. For example, you can type `status = $sndopr` then press Ctrl/E to expand it to the following text:

```
status = $sndopr (
             msgbuf = {~msgbuf~},
             [~chan   = {~chan~}~])
```

When you define a routine, LSE automatically generates the necessary NEW TOKEN or NEW PLACEHOLDER command to produce the templates for the given procedure call. Thus, you can expand and unexpand routines in the same manner as tokens. The following routine definition example shows how to cause LSE to generate the command, NEW TOKEN SYS$ADD_HOLDER, with the appropriate body for the current language:

```
PLSE NEW ROUTINE SYS$ADD_HOLDER SYSTEM_SERVICES
PLSE SET ROUTINE DESCRIPTION "    Add Holder Record To The Rights Database" -
       SYS$ADD_HOLDER SYSTEM_SERVICES
PLSE SET ROUTINE HELP TOPIC "System_Services $ADD_HOLDER" SYS$ADD_HOLDER SYSTEM_SERVICES
   PLSE NEW ROUTINE PARAMETER ID.V          SYS$ADD_HOLDER SYSTEM_SERVICES
   PLSE SET ROUTINE PARAMETER REQUIRED ID.V SYS$ADD_HOLDER SYSTEM_SERVICES
   PLSE SET ROUTINE PARAMETER VALUE ID.V    SYS$ADD_HOLDER SYSTEM_SERVICES

   PLSE NEW ROUTINE PARAMETER HOLDER.R           SYS$ADD_HOLDER SYSTEM_SERVICES
   PLSE SET ROUTINE PARAMETER REQUIRED HOLDER.R  SYS$ADD_HOLDER SYSTEM_SERVICES
   PLSE SET ROUTINE PARAMETER REFERENCE HOLDER.R SYS$ADD_HOLDER SYSTEM_SERVICES

   PLSE NEW ROUTINE PARAMETER ATTRIB.V           SYS$ADD_HOLDER SYSTEM_SERVICES
   PLSE SET ROUTINE PARAMETER OPTIONAL ATTRIB.V  SYS$ADD_HOLDER SYSTEM_SERVICES
   PLSE SET ROUTINE PARAMETER VALUE ATTRIB.V     SYS$ADD_HOLDER SYSTEM_SERVICES
```

You can access online help for any of the system services in any programming language. If you want help on any routine, place the cursor on the routine name and press PF1-PF2. The HELP entry for the system service contains information about the parameters.

If you want to view the contents of a given package or routine, use the SHOW command. If you want to modify the definitions of a package or routine, use the EXTRACT command.

Index