2

RL-TR-91-274, Vol Vb (of five)
Final Technical Report
November 1991

PENELOPE: AN ADA VERIFICATION ENVIRONMENT, Penelope User Guide: Guide to the Penelope Editor

ORA Corporation

Sponsored by
Strategic Defense Initiative Office

Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

APPROVED:

John C. Faust

JOHN C. FAUST
Project Engineer

FOR THE COMMANDER:

[illegible]

RAYMOND P. URTZ, JR.
Director
Command, Control and Communications Directorate

PENELOPE: AN ADA VERIFICATION ENVIRONMENT, Penelope User Guide: Guide to the Penelope Editor

C. Douglas Harper

Contractor:	ORA Corporation
Contract Number:	F30602-86-C-0071
Effective Date of Contract:	19 Aug 86
Contract Expiration Date:	30 Sep 89
Short Title of Work:	PENELOPE: AN ADA VERIFICATION ENVIRONMENT, Penelope User Guide: Guide to the Penelope Editor
Period of Work Covered:	Aug 86 - Aug 89
Principal Investigator:	Maureen Stillman
Phone:	(607) 277-2020
RL Project Engineer:	John C. Faust
Phone:	(315) 330-3241

This research was supported by the Strategic Defense Initiative Office of the Department of Defense and was monitored by John C. Faust, RL/C3AB, Griffiss AFB 13441-5700, under contract F30602-86-C-0071.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE November 1991	3. REPORT TYPE AND DATES COVERED Final Aug 86 - Aug 89

4. TITLE AND SUBTITLE PENELOPE: AN ADA VERIFICATION ENVIRONMENT, Penelope User Guide: Guide to the Penelope Editor	5. FUNDING NUMBERS C - F30602-86-C-0071 PE - 35167G/63223C PR - 1070/B413 TA - 01/03 WU - 02
6. AUTHOR(S) C. Douglas Harper	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) ORA Corporation 301A Dates Drive Ithaca NY 14850-1313	8. PERFORMING ORGANIZATION REPORT NUMBER ORA TR 17-9
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Strategic Defense Initiative Office, Office of the Secretary of Defense Wash DC 20301-7100 Rome Laboratory (C3AB) Griffiss AFB NY 13441-5700	10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-91-274, Vol Vb (of five)

11. SUPPLEMENTARY NOTES

RL Project Engineer: John C. Faust/C3AB/(315) 330-3241

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

We present an introduction to using the Penelope verification environment under X windows (v.10). The user is led step by step through the verification of a simple program, including the steps of invoking Penelope, editing a program, and using the commands in Penelope.

14. SUBJECT TERMS Ada, Larch, Larch/Ada, Formal Methods, Formal Specification, Program Verification, Predicate Transformers, Ada Verification	15. NUMBER OF PAGES 106
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Contents

List of Figures

1 Introduction

Penelope is a tool for writing verified Ada programs. With it, you can interactively produce Ada code, annotate the code in the Larch/Ada annotation language, and prove that the code correctly reflects its annotations.

To use the editor, you will need to know about:

- formal verification of programs [Gries 81];
- the Ada programming language [Ada 83];
- first-order logic [Kleene 67] and Larch/Ada [ORA 87a,ORA 87b,ORA 88].

Our approach to program verification is similar to Gries's. Because Ada is more complex than Gries's small, Pascal-like language, there naturally are differences, but they are outweighed by the similarities.

Larch/Ada is a straightforward first-order language used for annotating Ada. If you are comfortable with proofs involving formulas like

$$(\forall x)(x \neq 0 \Rightarrow (\exists y)(y = 1/x))$$

you will be comfortable with Larch/Ada. Read the references, if you have not already done so, then come back to this guide.

Penelope is a syntax-directed editor. You begin with a placeholder on your screen for a compilation_unit, which you expand according to the rules in the Ada reference manual. Each expansion leads you closer to a completed Ada program. As you guide the editor through the choices available in the rules, it generates the syntax. At the end, when you have expanded each identifier placeholder by typing in the corresponding identifier, you have a syntactically correct Ada compilation unit.

Almost all of this may be done by selecting items from menus with the mouse; very little typing is required. For instance, when you select a `BEGIN...END` construct, the '`BEGIN`' and '`END`' are generated for you, and a statement placeholder is generated between then. (Among other things, this means that you can never get a syntax error because of a missing '`END`'.)

The editor is a UNIX (tm) application program designed to run under X windows [Gettys 86] on a Sun work station. Its inputs are from the keyboard, the mouse, and from files edited earlier. Its outputs are to the screen and to files of your choice.

Because of the versatility of the X window system, I will assume that your Sun is configured exactly as are ours at Odyssey; otherwise, this guide would have to be too general to contain any specifics. So that this assumption is justified, we will provide all the files you need to install the editor.

The goal of editing is to produce verified compilable Ada units. As you save each verified unit in a file, it becomes available for two uses:

- Compilation. After you build up a good library of verified components, you will be able to assemble a large number of different verified programs.
- Re-editing. You may change and replace a unit, or use parts of old units in building a new one. (In the second case, you do not affect the originals.)

Penelope was created by use of the Synthesizer Generator (SG). Although this guide is self-contained, you may want to refer to the Synthesizer Generator Reference Manual [Reps 87], it and this guide being somewhat complementary. You will find the specifics of Penelope here, whereas you will find the general concepts underlying all SG editors in the Reference Manual. All the commands of the editor are standard SG commands: their descriptions in this guide are directed toward our editor; a full abstract description of these commands, applicable in any SG editor, is found in the SG Reference Manual.

It is also helpful, but not necessary, to be familiar with the emacs editor. Many of the commands of the editor have the same functions as the corresponding commands in emacs.

This guide is tutorial in nature. You should now sit down on a Sun work station and turn to Section 2.

2 Operating system basics

2.1 Logging into the Sun

Sit down at your idle Sun workstation, which should be giving you an invitation to log in, something like

Hi! I'm nestor, and I'll be your computer today.

(Whatever your installation decides to make this invitation say, it will include the name of your Sun. Everywhere you see the name 'nestor' in this guide, mentally replace it with the name of your Sun.)

Type

```
^C
```

That is, press the control key, and while holding it down, press 'c', then release both keys. (The caret '^' stands for 'control'. In the future, this notation will be used with several other letters; e.g., '^N'.) A few seconds will pass and you will be prompted with

nestor login: █

At this point, enter

⟨*your login name*⟩ ⟨*return*⟩

That is, type in your login name and follow it with a carriage return. If you do not know your login name, see your system administrator. After you enter your login name, nestor will give you the prompt

Password: █

Enter

⟨*your password*⟩ ⟨*return*⟩

After a few seconds pass, you should get a banner of one or more lines telling you that you have logged in.

2.2 Starting X windows

After the login banner is displayed, the Sun will ask you

run windows? [nxs] █

Enter

x ⟨*return*⟩

(for 'X windows'). You will see the screen go blank, and then you will see labels and boxes being drawn. The important one for now is the one that will be drawn last of all, in the lower left-hand corner. This box is an X window, the first one you will use. After a few seconds, this window will contain the prompt

```
label window? [nestor:ttyp0,n] ▯
```

Notice that the block following the prompt is hollow, unlike all the blocks that came before, which were solid. The solid blocks meant that the computer was ready to accept your answers to the prompts; the hollow block means that it is unready.

To make the block turn solid, use the mouse attached to your Sun. Slide the mouse on the pad and notice that the large X on your screen slides around as you do so. Move the X into the top of the X window, and two things will happen. First of all, the X will turn into an arrow. (It will turn into an elongated letter I if you move it down into the middle of the window. Either way will do.) Second, the block will turn solid. You have selected the X window; it will accept keyboard input. You should now see

```
label window? [nestor:ttyp0,n] █
```

The window is asking you whether or not you want its banner strip labeled with '**nestor:ttyp0**', as opposed to '**nestor:0**'. This is purely a matter of taste. If you want the longer label, just hit the return key. If you do not, enter the letter 'n' (no return is needed). Let us say that you enter

```
n
```

The next prompt you get will be

```
nestor: 1 % █
```

This is the window's general prompt, meaning that it is ready to accept commands. In particular it is ready for you to start up the editor.

2.3 Starting the editor

Enter

```
vcgen.X ⟨return⟩
```

At first it will seem that nothing is happening, but then the editor will draw a new window in the center of your screen, partly covering up your first window. Use the mouse to move the arrow into the new window, then go to Section 4.

3 Terminating your session

Once you are done using Penelope, you can exit X windows and log out of the Sun by using the X menu at the top and left of your screen. Place the mouse arrow on nestor, press the left button and hold it down. A menu will pop up with several entries on it. Still holding the left button down, move the mouse arrow to the entry '**Exit**'. When it is displayed in reverse video, release the left button. The system will then terminate both X windows and your Sun session.

4 Using the Editor

4.1 Demonstration: editing a simple subprogram

This subsection demonstrates the use of the editor to produce a simple, unverified Ada function, `multiply`. Verification conditions will be discussed later. Verification itself is discussed in the Penelope Tutorial [Hird 89].

After you invoke the editor (see Section 2), you should get a new window, the main edit window. The reverse video strip across the top containing the label '**main**' is the *title bar*. Below that is the *command line* (discussed in Appendix A). Ignore both these for now. Also ignore for now the *scroll bar* with its several arrows at the right of the window.

Below the command line is the largest pane in the window, the *object pane*, in which the text of your program appears. At the moment, it should look like

Figure 1:

```
<compilation unit>
```

The reverse video field (as simulated above by text in an enclosing box) is your *currently selected buffer* or *selection*, whose contents you are able to change. What is in your text buffer now is the starting placeholder for editing your Ada compilation unit. You could type in an entire compilation unit at this point, but there is a better way.

At the bottom of the editor window is the *help pane* containing a menu listing possible replacements for the placeholder.

Click on

`func-body`

with the left mouse button.

The mouse arrow will temporarily change shape to an hourglass (signifying that some time will pass: it is busy). Then your window should change to look like Figure 2 on the next page.

Figure 2:

```
--> 1 VCs NOT SHOWN!

 --| TRAIT <identifier> IS
 --| AXIOMS:
 --| END AXIONS;
 --| LEMMAS:
 --| END LEMMAS;

FUNCTION <designator>(<identifier> : <identifier>) RETURN <identifier>
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! []

IS


BEGIN
  <statement>
END ;
```

If your window does not come to look like Figure 2, chances are that your aim with the mouse was off a bit. The simplest fix to explain right now is to kill the editor and start over. Move the arrow out of the editor window back into the main X window (the one from which you invoked the editor), and type

`^C`

This kills the editor. Enter

`vcgen.X` ⟨*return*⟩

to restart.

If your window does look like Figure 2, all is well. Ignore the warning about VCs for now; they will be discussed in Section 6. Note that a lot of the Ada has been generated for you, but that there are a number of placeholders to be resolved before the whole thing is legal Ada. What you are seeing is a template for an Ada function, whose "blanks" you will "fill in" as you go along, either with text or with other, more specific, templates.

Note also that the lines

```
 --| TRAIT <identifier> IS
 --| AXIOMS:
 --| END AXIOMS;
 --| LEMMAS:
 --| END LEMMAS;
```

are in reverse video, indicating that the multiline field is the current selection. We do not discuss Larch Traits here; what we want to talk about now is the Ada code.

The first thing to do is to name the Ada function. To do this, click on the placeholder

`<designator>`

after the keyword '**FUNCTION**'. Your screen should change to look like the next figure.

Figure 3:

```
--> 1 VCs NOT SHOWN!

FUNCTION <designator> (<identifier> :  <identifier>) RETURN <identifier>

  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS


BEGIN
  <statement>
END ;
```

Notice that the Larch trait has disappeared. You can always get it back, but do not worry about it now.

The current selection is for the function name. Enter

`multiply` ⟨*return*⟩

After you hit the return key, your screen should change to look like the next figure.

Figure 4:

```
--> 1 VCs NOT SHOWN!

FUNCTION multiply (<identifier> :  <identifier>) RETURN <identifier>

  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS


BEGIN
  <statement>
END multiply;
```

Notice that the text buffer contains parentheses, two identifier placeholders, and a colon, comprising the starting text for the function's formal_part. This entire field, instead of just the first '`<identifier>`', is your current text buffer

You have several options at this point.

- You could type in

  ␣ ⟨*return*⟩

  (that half-box stands for a blank space) to remove the field entirely. While this option has to be present, since the Ada RM allows a function to have no formal_part, it is not what you want to do for this demonstration. For now, just note that this is how to get entirely rid of the unwanted text in a buffer.

- You could type in what is to be the entire formal_part right now by entering

  `(m,n:integer)`

  (no ⟨*return*⟩) and then selecting the

  `<identifier>`

  after '**RETURN**', which would take you directly to Figure 12. There are two main drawbacks to doing this. If you make a typing mistake, you will have to get out of it by killing the editor. Also, you will not learn as much about the editor by doing this as you will by doing things the long way. Instead:

- Hit

  ⟨*return*⟩

  to tell the editor to move to the next text buffer. Since you have not finished expanding the formal_part to Ada code, it should come as no surprise that the next text buffer is a sub-buffer of the current one.

Figure 5:

```
--> 1 VCs NOT SHOWN!

FUNCTION multiply(<identifier> :  <identifier>) RETURN <identifier>

  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS


BEGIN
  <statement>
END multiply;
```

The text buffer now contains both of the identifier placeholders and the colon, but not the parentheses; you are building the formal_part as a list of one or more parameter_specifications. What you see is a template for the first one in the list. Again, you have options:

- You could type the entire specification without hitting ⟨*return*⟩:

  `m,n:integer`

  and instead click on

  `<identifier>`

  after the '`RETURN`' on the same line. You would then be seeing a screen like Figure 12.

  You can always use the mouse to select buffers, which not only gives you the freedom to skip around in the program at will, but also often saves you intermediate steps. In this case, it would save you from stepping through Figures 6-11.

- Do it the long way this time. Hit

  ⟨*return*⟩

  to select the sub-buffer for the first identifier.

Figure 6:

```
--> 1 VCs NOT SHOWN!

FUNCTION multiply( <identifier>  :  <identifier>) RETURN <identifier>

  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS


BEGIN
  <statement>
END multiply;
```

The text buffer is for a list of parameters to the function, all of a type to be named later. You have options.

- (Recommended) Enter only the first parameter:

  m ⟨*return*⟩

  and go to Figure 7.

- Enter both parameters at once:

  m,n ⟨*return*⟩

  and go to Figure 8.

Figure 7:

```
--> 1 VCs NOT SHOWN!

FUNCTION multiply(m, <identifier>:  <identifier>) RETURN <identifier>

  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS

BEGIN
  <statement>
END multiply;
```

The text buffer is for for the next parameter in the parameter list, if any. Enter

n ⟨*return*⟩

for the second parameter.

Figure 8:

```
--> 1 VCs NOT SHOWN!

FUNCTION multiply(m, n, <identifier>:  <identifier>) RETURN <identifier>

  --> GLOBAL ();
  --! WHERE * * *
  --! END WHERE;
  --! VC Status: hidden
  --! □

IS

BEGIN
  <statement>
END multiply;
```

Again, the text buffer is for the next parameter, if any. This time, there is not one, so hit

⟨*return*⟩

Figure 9:

```
--> 1 VCs NOT SHOWN!

FUNCTION multiply(m, n : IN <identifier>) RETURN <identifier>

  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS


BEGIN
  <statement>
END multiply;
```

Notice that now that you have ended this list of parameters and are concerned with their typemark, the editor is smart enough to remove the comma from after the 'n'. Notice also that the default mode mark '**IN**' has been generated after the colon, and is the content of the current buffer. The editor automatically generates the defaults as you go, but you need not accept them. Once a default value is displayed in a buffer, you may edit that buffer.

You have options.

- You could select the mode you want from the menu

  `in   out   in-out`

  by clicking with the mouse. You would want to do this only in the case of a procedure, since all parameters to functions must have mode **IN**. (In a later version of the editor, the two sorts of subprogram formal_part will be handled separately, and you will not have this spurious choice for functions.)

- You could enter the mode as text.

- You could accept the default by selecting another buffer. This is what you want to do, so hit

  ⟨*return*⟩

  and see the next figure.

Figure 10:

```
--> 1 VCs NOT SHOWN!

FUNCTION multiply(m, n :  <identifier> ) RETURN <identifier>

  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS


BEGIN
  <statement>
END multiply;
```

Notice that the '**IN**' has disappeared. The editor will not leave Ada default values explicitly displayed. The only time one shows up is when its buffer is selected, enabling you to modify it if you so desire.

The text buffer is for the type_mark of this parameter_specification. Enter

integer ⟨*return*⟩

to finish the specification.

Figure 11:

```
--> 1 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer;

<identifier> :  <identifier> ) RETURN <identifier>

  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS


BEGIN
  <statement>
END multiply;
```

The Ada formal_part consists of one or more instances of parameter_specification, separated by semicolons. If you wanted another parameter_specification, you would proceed just as you did for the first one, filling in the blanks as you went along. You have, however, given all the parameters that `multiply` takes, so skip over this buffer by clicking on the

`<identifier>`

after the 'RETURN'.

You could, if you wanted to take the long way, keep hitting ⟨*return*⟩ to wade through all the optional buffers until you reached the next figure. You can *always* move forward in editing by hitting ⟨*return*⟩, but it is the longest possible way to do so.

Rule of thumb: use ⟨*return*⟩ when you want to select the very next bu
use the mouse to select more distant buffers.

Figure 12:

```
--> 1 VCs NOT SHOWN!

FUNCTION multiply(m, n :  integer) RETURN <identifier>

  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS


BEGIN
  <statement>
END multiply;
```

Notice that the buffer of Figure 11 has completely disappeared. This is because the second parameter_specification was optional. By skipping over it, you indicated that you did not want it. Enter

integer ⟨*return*⟩

for the identifier.

Figure 13:

```
--> 1 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();

--| WHERE * * *
--| END WHERE;

  --! VC Status: hidden
  --! □

IS

BEGIN
  <statement>
END multiply;
```

The multiline text buffer is for the entry/exit annotations of `multiply`, which I will defer for now. Note that so far as Ada is concerned, the buffer contains comments. This demonstration is concerned solely with Ada editing; a discussion of editing annotations can be found below in Section 5. Skip over this buffer by hitting

⟨*return*⟩

Figure 14:

```
--> 1 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;

 --!  VC Status:  hidden
 --!  □

IS


BEGIN
  <statement>
END multiply;
```

The multiline text buffer is for the proof of the VC for `multiply`. VCs are discussed in Section 6. Skip over this buffer by hitting

⟨*return*⟩

Figure 15:

```
--> 1 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS

   <basic declarative item>

BEGIN
  <statement>
END multiply;
```

Penelope has generated the placeholder

`<basic declarative item>`

for you. At this point, you will declare your local variables. Before you do so, however, recall that there are two different kinds of declarative items, basic and later. You should be aware, for instance, that the bodies of subprograms may be developed only from later_declarative_item placeholders. For more detail, see Chapter 3 (especially Section 3.9) of the Ada Reference Manual [Ada 83].

Click on menu item

`object`

to set up the template for an object declaration. This is the first case in which you are selecting a more detailed template rather than filling in text, so you may want to look carefully at the next figure.

Figure 16:

```
--> 1 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS

  <identifier> : <identifier> := <exp>;

BEGIN
  <statement>
END multiply;
```

The text buffer is for the identifier_list of the object_declaration, which you will expand to an actual list of identifiers. Again, only the first identifier is required; others are optional. Type

prod

(do not hit ⟨*return*⟩). If you hit ⟨*return*⟩ now, your screen would look like Figure 17 below, and you would have to hit ⟨*return*⟩ or click the mouse to skip over the optional field to get the screen to look like Figure 18, which is what you really want.

The shortcut to Figure 18 is to use the general method to keep from generating optional fields after the one you are editing. Enter the `forward-sibling` command,

`ESC-^N`

Let me revise what I said earlier about moving to new buffers.

Rule of thumb: use ⟨*return*⟩ to move to the very next buffer, optional or not (optionals will be generated); use `ESC-^N` to select the very next displayed buffer (optionals will not be generated); use the mouse to move to any displayed buffer.

Figure 17:

```
--> 1 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS

  prod, <identifier> :  <identifier> := <exp>;

BEGIN
  <statement>
END multiply;
```

This is not what your screen will look like if you did the last step correctly. You want the next figure.

If your screen does look like this, hit

⟨*return*⟩

to get it to look like the next figure.

Figure 18:

```
--> 1 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS

  prod : <identifier> := <exp>;

BEGIN
  <statement>
END multiply;
```

Enter

`integer` ⟨*return*⟩

for the type_mark.

Figure 19:

```
--> 1 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS

  prod :  integer  := <exp> ;

BEGIN
  <statement>
END multiply;
```

This buffer includes both the '`:=`' and the placeholder '**`<exp>`**'. You may declare an Ada object without initializing it: this buffer is selectable so that you can blank out the initialization, if you wish.

In this case, you do want to initialize the object, so select the expression placeholder by hitting

⟨*return*⟩

and turn to the next page.

Figure 20:

```
--> 1 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS

  prod :  integer := <exp>;

BEGIN
  <statement>
END multiply;
```

Type

0 ⟨*return*⟩

and turn to the next figure.

Figure 21:

```
--> 1 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;

  <basic declarative item>

BEGIN
  <statement>
END multiply;
```

Now use the same methods as before to get your screen to look like the following.

Figure 22:

```
--> 1 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;

  mtemp :  integer := <exp> ;

BEGIN
  <statement>
END multiply;
```

Type

m

for the initial value, but do not hit ⟨*return*⟩, which would uselessly generate another '**<basic declarative item>**'; Instead, use the mouse to select the buffer for

<statement>

and see Figure 23.

Figure 23:

```
--> 1 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN

  <statement>

END multiply;
```

Choose

`while-loop`

from the menu, and turn the page.

Figure 24:

```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN

 --!  VC Status:  hidden
 --!  □

  WHILE <exp> LOOP
    --|INVARIANT = <term>;
    <statement>
  END LOOP;
END multiply;
```

The selection contains a warning from the editor that the verification condition newly generated for the loop is not displayed. Since I am deferring VCs for now, click on

<exp>

one line down, and turn the page.

Figure 25:

```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □

  WHILE <exp> LOOP

    --|INVARIANT = <term>;
    <statement>
  END LOOP;
END multiply;
```

Type

mtemp>0

(no ⟨*return*⟩)

and select the placeholder

<statement>

in the body of the loop. Ignore for now the invariant of the loop, which will be discussed in Section 5. Turn to the next figure.

Figure 26:

```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP
    --|INVARIANT = <term>;

    <statement>

  END LOOP;
END multiply;
```

Select

`assignment`

from the menu, and turn the page.

Figure 27:

```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP
    --|INVARIANT = <term>;

    <name>:=<exp>;

  END LOOP;
END multiply;
```

Enter

prod ⟨*return*⟩

Since there are no optionals after the identifier and before the '`:=`', hitting ⟨*return*⟩ takes you right to '`<exp>`'. See the next figure.

Figure 28:

```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP
    --|INVARIANT = <term>;

    prod:=<exp>;

  END LOOP;
END multiply;
```

Enter

prod+n ⟨*return*⟩

and turn the page.

Figure 29:

```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP
    --|INVARIANT = <term>;
    prod:=(prod+n);

    <statement>

  END LOOP;
END multiply;
```

Notice that parentheses have been generated for you. The editor accepts both parenthesized and unparenthesized input, and gives fully parenthesized output.

The body of the loop is formally a list with optionals, so hitting ⟨*return*⟩ generated another '`<statement>`' and selected it. Fill this placeholder with another assignment statement, but try it a different way this time. Instead of selecting from the menu, enter

`mtemp:=mtemp-1;` ⟨*return*⟩

The syntax must be completely correct. If not, the editor will print '`syntax error`' on the command line. You will then either have to use the techniques of Section 4.3.2, or kill and restart Penelope in the manner described in the discussion of Figure 2.

Go to the next page.

Figure 30:

```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP
    --|INVARIANT = <term>;
    prod:=(prod+n);
    mtemp:=(mtemp-1);

    <statement>

  END LOOP;
END multiply;
```

You have completed the body of the loop. To move beyond it, hit

⟨*return*⟩

and turn the page.

Figure 31:

```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP
    --|INVARIANT = <term>;
    prod:=(prod+n);
    mtemp:=(mtemp-1);
  END LOOP;

  <statement>

END multiply;
```

You now may place a statement after the loop. Select

`return`

from the menu, and see the next page.

Figure 32:

```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP
    --|INVARIANT = <term>;
    prod:=(prod+n);
    mtemp:=(mtemp-1);
  END LOOP;

  RETURN <exp>;

END multiply;
```

Type

prod

(no ⟨*return*⟩)

for the returned expression. Since this is the last Ada statement in the program, you do not want to generate another placeholder by hitting ⟨*return*⟩. Instead, hit

ESC-^N

to avoid generating optionals and turn the page to see your completed, syntactically correct Ada program.

Figure 33:

```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP
    --|INVARIANT = <term>;
    prod:=(prod+n);
    mtemp:=(mtemp-1);
  END LOOP;

  RETURN prod;

END multiply;
```

To learn how to save your file, turn to Subsection 4.2.

4.2 Input and Output

4.2.1 Output: writing files

Let us continue the demonstration a bit further. Now that you have a completed Ada compilation unit, you want to write it. The way to do this is to enter the `write-named-file` command,

`^X^W`

and wait for a new window to open. Notice that this window has

`<filename>`

selected. You may choose any name you want, but for this demonstration, enter

`multiply.text` ⟨*return*⟩

See that the next buffer has been selected:

$\boxed{\text{structure}}$

Down in the menu, you will see:

`text structure`

These represent the two formats available for output: 'text' is for ordinary makes-sense-to-humans Ada; 'structure' is for the internal editor representation of the file. Use the mouse to select

`text`

You are now ready to write the file. Click on the

`Start`

in the upper right-hand corner of the window to execute **`write-named-file`**.

When **`write-named-file`** is complete, you will be returned to the main edit window, whose command line will contain the message

`wrote file multiply.text`

You will then have written an ASCII version of **`multiply`**, suitable for compiling.

To write a structure-format copy, enter

`^X^W`

again, and note that the name field has the old value you typed:

$\boxed{\texttt{multiply.text}}$

You must change this so that it contains '**`multiply.structure`**'. To do so, use the **`delete-selection`** command,

`^K`

to remove the contents of the selected buffer and bring back the placeholder. (This works in general. For more information, see Subsubsection 4.3.2.)

You now have

$\boxed{\texttt{<filename>}}$

selected. Type

`multiply.structure`

and then select

`Start`

just as before. When you return to the main editor window, the editor will inform you in the command line that you have written a structure-format version of the file.

4.2.2 Interlude: looking at your output

Now that your files are written, leave the editor and have a look at them. You already know the rude way to leave the editor: go to the X window from which you started it and kill it. The polite way to leave the editor is by typing the `exit` command,

`^X^C`

To see the text file, enter

more multiply.text ⟨*return*⟩

at the X window prompt. Note that the file is just as it should be, ready for the compiler.

Now turn to Subsubsection 4.2.3 to learn how to present your files as input to the editor.

4.2.3 Input: reading in your files

There are two ways to read files into the editor. The first is from the X window prompt as you start the editor, and the second is from within the editor itself. The first way allows you to edit a file that already exists; the second allows you to read a file into a text buffer.

To edit a file that already exists, first be in an X window with a solid box prompt. Then start the editor, just as before, but this time enter the name of the file after the '**vcgen.X**'. For instance, let us say that you want to edit `multiply.text`. Enter

vcgen.X multiply.text ⟨*return*⟩

and look at the screen. It looks just like Figure 33, except that the entire file stands out in reverse video against the rest of the window, and the command line at the top of the window says '`Syntax error in line 17`'. Why?

The reason that you have a syntax error is that your input contains placeholders in the Larch/Ada text. (Notice the error marker, an elongated letter 'I', situated just inside the placeholder.) The editor will not accept placeholders anywhere in text input, not even in Ada comments. Furthermore, you cannot select away from the buffer with a syntax error: the editor makes you fix syntax errors before you do any more editing.

You could go to Subsubsection 4.3.2, and learn the long, grubby way to fix this, but this is not necessary. Instead, abandon this edit, and then edit the structure file. Type

`^X^C`

to leave the editor, and then restart it with the structure file as input by typing

vcgen.X multiply.structure ⟨*return*⟩

Your main edit window will come to look as it did a minute ago, but with a different command line from the last time.

Placeholders in structure files are accepted: the command line tells you '`Read multiply.structure`' instead of '`Syntax error`'; your window looks essentially the same as Figure 33 (for a discussion of what to do about all that reverse video, see Subsection 4.3). By writing and then re-editing structure files, you may suspend and then resume editing whenever you like.

Rule of thumb: use structure files for intermediate versions; use text files for the final, compiler-ready versions.

Now let us look at the other way to read files into the editor. Leave the editor by typing

```
^X^C
```

Then bring up the editor without a filename argument. Enter

vcgen.X ⟨*return*⟩

Things will look just like Figure 1 again. This time, though, do not use the menu. Instead, en.er the `read-file` command,

```
^X^R
```

A new window will open up, very much like the one above for writing. Type

```
multiply.structure
```

and click the

```
Start
```

box. This will replace the entire main window contents with the text represented in the file multiply.structure. Your sereen will look just as it did after you typed '**vcgen.X multiply.structure**'.

(Of course, you could have read **multiply.text** instead, but you still would have been told '`Syntax error`'.)

Do not leave the editor. The demonstration continues in Subsection 4.3.

(For the advanced aspects of reading and writing, you should see the Synthesizer Generator Reference Manual [Reps 87].)

4.3 Changing the file

Here I treat three related topics: correcting syntax errors; revising syntactically correct files; and selecting buffers. I consider the third topic together with the first two because Penelope is a structure editor, not a text editor. To correct syntax errors, you must know more than has yet been said about text buffers; to revise a file, you must know which buffers to select for changing.

4.3.1 Buffer selection

I am assuming that you have the editor the way you left it in Subsection 4.2. If you do not, you should get it that way by typing

vcgen.X multiply.structure ⟨*return*⟩

to your X window prompt.

The buffer structure of the editor reflects the tree structure of the Ada (and Larch/Ada) language(s). There is a master text buffer for the entire file. Every other buffer for the file is a subtree of the master buffer. This explains why the entire file is in reverse video: the entire compilation_unit buffer was replaced by the action of the `read-file`, and the entire buffer was still selected afterwards. The same thing happens when you type text into a buffer: the buffer is still selected until you hit ⟨*return*⟩, use the mouse, or otherwise select another buffer.

As you already know, every placeholder has a corresponding buffer. Often when that buffer is edited, other buffers are created, sub-buffers of the first. There are more buffers in the editor than have been introduced so far. To see every buffer in the file, repeatedly hit

⟨*return*⟩

Penelope will reveal all the buffers in the file in preorder, even the ones normally hidden from view. It will also generate optionals as you go along, then suppress them after you pass them by.

The first ⟨*return*⟩ causes Penelope to generate a Larch trait. The second ⟨*return*⟩ selects the buffer for the name of the trait. The third generates an `INTRODUCES` statement. The fourth selects the name of the function to be introduced. At the twelfth ⟨*return*⟩, you have completed the traversal of the Larch trait subtree. Since you have entered no text, Penelope supresses the display of the trait.

The twelfth ⟨*return*⟩ also generates

```
<compilation unit>
```

between the warning about missing VCs and the rest of the file. The next ⟨*return*⟩ suppresses it, and selects the rest of the file. This pattern of generating optionals before and after visible text continues all through the file.

When the end of the function is reached, a new

```
<compilation unit>
```

is generated after the function body. Hitting ⟨*return*⟩ one last time will suppress it, and take you full circle back to where you started, with the master buffer selected.

Note that there were three buffers associated with

```
(mtemp>0)
```

in the `WHILE` statement: one for the entire expression, parentheses and all; one for the subexpression '`mtemp`'; and one for the subexpression '`0`'. Selecting the latter two would be easy: point to either and click. Select the entire expression by clicking on either of the parentheses or on the '>'. This should make intuitive sense. They belong to the expression, but to neither subexpression. This generalizes.

Rule of thumb: Clicking on a symbol in the file selects the buffer for the smallest structure containing that symbol.

There is a consequence of this that would appear strange if you did not know the rule of thumb. As you move down through the file, you will come to a screen looking like the figure on the next page.

Figure 34:

```
--> 2 VCs NOT SHOWN!

FUNCTION multiply(m, n, <identifier> :  integer) RETURN integer

  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP
    --|INVARIANT = <term>;
    prod:=(prod+n);
    mtemp:=(mtemp-1);
  END LOOP;
  RETURN  prod;
END multiply;
```

This time, instead of typing ⟨*return*⟩, use the mouse to select the '**integer**' in the formal_part. You might expect the line to change to look like

```
FUNCTION multiply(m, n :   integer ) RETURN integer
```

---

but it does not. Instead, it comes to look like

```
FUNCTION multiply(m, n :  integer) RETURN integer
```

This is because the optional buffer for the '**<identifier>**' disappears as you select away from it. The smallest structure containing the text '**integer**' that persists through the change effected by clicking is the entire function header buffer. (The formal_part buffer does not count because the number of its sub-buffers changes as you click.)

So far you have seen buffer selection by means of the mouse and the two commands ⟨*return*⟩ and ESC-^N. While you can get to any buffer you want with these, you probably would like to know some shortcuts. I will introduce two more methods here. There are many other commands for moving around in the file: Chapter 3 of the Synthesizer Generator Reference Manual [Reps 87] is exhaustive.

To get to the master buffer for the file, use the **`beginning-of-file`** command,

`ESC-<`

To get to the last buffer in the file, use the **`end-of-file`** command,

`ESC->`

Now what if your file is so long that it cannot be displayed all at once on one screen, and the buffer you want is not in sight? You can still move forward by means of ⟨*return*⟩ or `ESC-^N`, and eventually get to it, if it is ahead of you. If it is behind you, you can use the clumsy expedient of going to the beginning of the file, and then moving forward. This will always work.

The direct way to select a previous buffer is by means of the **`backward-preorder`** command

`^P`

which has roughly the opposite effect from ⟨*return*⟩.

The scroll bar at the right edge of the main edit window enables more rapid progress.

- Click on the upward arrow-and-rectangle in the upper right of your screen to scroll to the top of the file.
- Click on the triple upward arrow to scroll up one screen display.
- Click on the double upward arrow to scroll up half a screen display.
- Click on the single upward arrow to scroll up one line.
- The downward arrows have the obvious corresponding uses.

Notice I said 'scroll', and not 'move'. Using the arrows does not change the buffer selection, so you have not moved within the file; you have just changed the display. Once you change the display, however, you are free to select any displayed buffer on the screen.

This should give you enough tools for rapidly selecting any buffer you want in the file.

4.3.2 Error correction

The editor insists upon syntactically correct input. When you type text into a buffer and try to select another buffer, the editor parses your input. If adding your text to the compilation unit causes a syntax error, an error message is generated on the command line: '`syntax error`'. A cursor will also be displayed, marking the current position in the selected buffer. You must correct the error in the selected buffer before doing anything else (except quitting). The editor will not allow you to defer fixing it while you edit another buffer. The buffer you edited remains selected until you fix the error in it or leave the editor. So far as the editor is concerned, you have not finished text entry until the text is syntactically correct.

In changing the buffer, You may move the cursor left or right, you may delete characters, and you may enter new characters. These methods for text editing work in any text buffer. I introduce them here, but you do not have to be recovering from a syntax error to use them.

Let us consider examples. Invoke the editor without a filename argument: this will be just a throw-away session. You do not want to edit an old file, and you will not want to save your file when you are done.

Get your screen to look like Figure 3, and select the

`<designator>`

placeholder with the mouse. Enter

`funky␣name` ⟨*return*⟩

which is syntactically incorrect because of the embedded space. Notice the I-beam cursor in the reverse video field, flagging the buffer in which the syntax error occurs.

Figure 35:

```
--> 1 VCs NOT SHOWN!

FUNCTION funky name (<identifier> :  <identifier>) RETURN <identifier>

  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS

BEGIN
  <statement>
END ;
```

To recover from this error, enter the **delete-selection** command,

^K

(no ⟨*return*⟩), which deletes the text in the current buffer, but leaves the buffer selected. Your screen will look come to look like Figure 36 below.

Figure 36:

```
--> 1 VCs NOT SHOWN!

FUNCTION ▌        (<identifier> :  <identifier>) RETURN <identifier>

  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS


BEGIN
  <statement>
END ;
```

The buffer is empty. (That thin solid box is the cursor, superimposed on the remains of the reverse-video field.) You may now enter text exactly as though *you had just selected* the buffer. Type

funkyname ⟨*return*⟩

leaving out the space and adding the ⟨*return*⟩. Your screen should come to look like Figure 37 below.

Figure 37:

```
--> 1 VCs NOT SHOWN!

FUNCTION funkyname (<identifier> : <identifier>) RETURN <identifier>

  --> GLOBAL ();
  --! WHERE * * *
  --! END WHERE;
  --! VC Status: hidden
  --! □

IS

BEGIN
  <statement>
END funkyname;
```

Using `^K` to empty the field for starting over is less brutal than restarting the editor, but sometimes you want an even lighter touch. Let us look at some other capabilities. Get your screen to look like Figure 35 again. This time, use the mouse to click on the space between '`funky`' and '`name`'. The line in question should now look like this:

```
FUNCTION funky name (<identifier> : <identifier>) RETURN <identifier>
```

If the cursor is to one side or the other, click again until it does look like the above. To remove the space, use the `delete-next-character` command,

`^D`

You should have:

```
FUNCTION funkyname (<identifier> : <identifier>) RETURN <identifier>
```

Enter

⟨*return*⟩

Your screen should look like Figure 37, just as before.

Select

`funkyname`

with the mouse. The line should now look like

FUNCTION $\boxed{\texttt{funkyname}}$ **RETURN** **<identifier>**

First notice that the *formal_part* is gone. You selected the entire optional field and then selected away from it, telling the editor that you did not want it.

Notice also that no cursor is displayed. You need to have a cursor on the screen for the following text editing commands to have effect, so select, say, the '`n`' in '`funkyname`' with the mouse and type a space,

␣

to get:

FUNCTION $\boxed{\texttt{funky }\vert\texttt{name}}$ **RETURN** **<identifier>**

Use the `delete-previous-character` command by hitting the delete key,

DEL

to get:

FUNCTION $\boxed{\texttt{funky}\vert\texttt{name}}$ **RETURN** **<identifier>**

This may seem arcane, but at lease you have a displayed cursor. I will explain why you have to do all this after you have had practice positioning the cursor.

Enter the `beginning-of-line` command,

^A

to get:

FUNCTION $\boxed{\vert\texttt{funkyname}}$ **RETURN** **<identifier>**

Notice that the only change is in the position of the cursor.

Enter the `end-of-line` command,

^E

to get the line to look like:

FUNCTION $\boxed{\texttt{funkyname}\vert}$ **RETURN** **<identifier>**

Enter the `left` command,

```
^B
```

('B' for 'back') to get:

```
FUNCTION funkynam|e RETURN <identifier>
```

Do it twice more to get:

```
FUNCTION funkyn|ame RETURN <identifier>
```

Use the `delete-next-character` command,

```
^D
```

to remove the 'a' and get:

```
FUNCTION funkyn|me RETURN <identifier>
```

Use the `erase-to-end-of-line` command,

```
ESC-d
```

to remove everything after the cursor:

```
FUNCTION funkyn| RETURN <identifier>
```

Use the `delete-previous-character` command,

```
DEL
```

to remove the final 'n' and get:

```
FUNCTION funky| RETURN <identifier>
```

Enter

```
^A
```

to get to the beginning again and use the `right` command

^F

('F' for 'forward') to get:

```
FUNCTION |f[unky| RETURN <identifier>
```

Do it twice more to get:

```
FUNCTION |fun[ky| RETURN <identifier>
```

Now use the `erase-to-beginning-of-line` command,

`ESC-DEL`

to remove everything before the cursor, and get:

```
FUNCTION |[ky| RETURN <identifier>
```

You may intersperse text entry with any of these commands. For instance, the outcome of now typing

^F e ^E

is:

```
FUNCTION |key[| RETURN <identifier>
```

This is almost everything you need to know about the techniques available for error correction. You should be aware, though, that you do not encounter syntax errors merely from typing mistakes. Recall that the editor does not accept placeholders in textual input. You will get one syntax error message for each placeholder in a text input file; you can use `^K` to remove the offending fields, or you can change them to whatever you like. The main thing is that you must bring the input file into syntactic correctness before you can proceed further.

Let us get back to this arcana about selecting a buffer, typing a space, and then deleting it. When you select a buffer that already has text, no cursor is displayed until you enter new text. This means that none of the cursor-positioning commands will have effect until you do enter text. It may seem obvious and hardly worth stating, but you cannot move a cursor around if it is not there to be moved. If you forget this, though, the above dubious feature of the editor will someday perplex and annoy you.

Let us look at an annoying example. Select away from

key

and then select it again with the mouse arrow on 'e'. No cursor appears; you merely have:

```
FUNCTION [key] RETURN <identifier>
```

The editor knows that you are positioned at the 'e', but it is not showing you a cursor, and it will not let you change the position without a cursor showing. That is why the following input:

`^E strokes`

which you want to use to produce:

```
FUNCTION [keystrokes] RETURN <identifier>
```

---

does not produce that effect. *What you get instead is:*

```
FUNCTION [kstrokesey] RETURN <identifier>
```

One more annoyance, then I will be done with this subsubsection and you can exit the editor: you would better not use the backspace key (or its keystroke alias, `^H`) during text entry. It is not what you might expect: it is not the **`left`** command; it is the **`backward-with-optionals`** command, and if you use it instead of `^B`, you will be rudely disappointed when the editor selects a previous buffer.

4.3.3 Textual revision

You will often wish to change portions of the program as you are editing it. Sometimes these changes will be minor, sometimes they will extend over several lines. By learning to exploit the buffer structure of the editor, you will find that revision can be easy. The basic move is to select and then modify the smallest buffer that contains the text to be changed.

The simplest way to modify a buffer is to use

```
^K
```

to delete the old contents, and then to enter the new text. You have seen this above. All the techniques of Subsubsection 4.3.2 are available to you for purposes of revision. An example will show you some aspects of the old techniques that might not have occurred to you, and will serve to introduce some new techniques.

This example is just for fun. The Ada program you develop will not make a great deal of sense, and you will not want to save it, but editing it will teach you about revision. Start the editor up with input file **multiply.structure** so that your screen looks like Figure 33 again (except that the whole file will be selected). You know that you could change the assignment statement

```
mtemp:=(mtemp-1);
```

to

```
mtemp:=(mtemp-2);
```

by selecting the '1', using `delete-selection` and typing '2'. Be a bit more ambitious. Select the entire loop by clicking on

```
LOOP
```

Your screen should now look just like Figure 33, selected buffer and all. Delete the entire loop with

```
^K
```

Your screen should come to look like Figure 38 below:

Figure 38:

```
--> 1 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN

<statement>

  RETURN  prod;
END multiply;
```

You can now edit the statement buffer. Select the menu item

`if-then-else`

so that your screen comes to look like Figure 39 below:

Figure 39:

```
--> 1 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN

  IF <exp> THEN

    <statement>
  ELSE
    <statement>
  END IF;
  RETURN  prod;
END multiply;
```

Type

m>n

in the expression buffer, and edit the first statement buffer so that your screen looks like Figure 40 below:

Figure 40:

```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  IF (m>n) THEN

     --!  VC Status:  hidden
     --!  □
     WHILE (mtemp>n) LOOP
       --|INVARIANT = <term>;
       mtemp:=(mtemp-1);
     END LOOP;

  ELSE
    <statement>
  END IF;
  RETURN  prod;
END multiply;
```

Use the `copy-to-clipped` command,

`ESC-^W`

to copy the selected buffer to the special CLIPPED buffer. Then select the statement buffer and use the `copy-from-clipped` command

`ESC-^Y`

to copy the CLIPPED buffer to the selected placeholder. Your screen should now look like Figure 41 below:

Figure 41:

```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  IF (m>n) THEN
    --! VC Status: hidden
    --! □
    WHILE (mtemp>n) LOOP
      --|INVARIANT = <term>;
      mtemp:=(mtemp-1);
    END LOOP;
  ELSE

    --!  VC Status:  hidden
    --!  □
    WHILE (mtemp>n) LOOP
      --|INVARIANT = <term>;
      mtemp:=(mtemp-1);
    END LOOP;

  END IF;
  RETURN  prod;
END multiply;
```

There are several commands similar to the above.

- The `cut-to-clipped` command, `^W`, copies from the current selection to CLIPPED, and deletes the selection.
- The `paste-from-clipped` command, `^Y`, copies from CLIPPED to the selected placeholder, and deletes the contents of CLIPPED. This command like `copy-from-clipped`, is sensitive to the structure of the program. The target buffer must be syntactically compatible with the original source of the CLIPPED contents. You have seen that you can copy a loop_statement into a statement placeholder. However, you cannot copy

  ```
  --| WHERE * * *
  --| END WHERE;
  ```

into a

`<statement>`

placeholder, for instance.

- The **`text-capture`** command (which has no keystroke form) offers a way around the limitations above. It copies the current selection *as text* into CLIPPED.

  You must use either the mouse or the command line to invoke this command.

- The `copy-text-from-clipped`, `ESC-^T`, copies the text previously captured in CLIPPED to the selected text buffer.

You should also exploit the menu in revision. The items **`insert-before`** and **`insert-after`** are especially helpful, creating appropriate buffers in the body of your program. In the second loop, select the statement

```
mtemp:=(mtemp-1);
```

and then the menu item

`insert-before`

to get your screen to look like Figure 42 below:

Figure 42:

```
--> 3 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE * * *
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  IF (mtemp>0) THEN
    --! VC Status: hidden
    --! □
    WHILE (mtemp>0) LOOP
      --|INVARIANT = <term>;
      prod:=(prod+n);
      mtemp:=(mtemp-1);
    END LOOP;
  ELSE
    --! VC Status: hidden
    --! □
    WHILE (mtemp>0) LOOP
      --|INVARIANT = <term>;
      prod:=(prod+n);

       ┌──────────────────┐
       │<statement>       │
       │mtemp:=(mtemp-1); │
       └──────────────────┘

    END LOOP;
  END IF;
  RETURN  prod;
END multiply;
```

The selected buffer now contains both the old assignment statement and the new statement placeholder. You will have to select the placeholder buffer in order to edit it separately.

This concludes the discussion of revision.

5 Annotations

Making Larch/Ada annotations is very similar to entering Ada text. Some annotations are entered into placeholders generated for you as optionals, such as those between '`--| WHILE`' and '`--| END WHILE;`'. Others, such as embedded assertions, are entered into statement buffers as Ada comments. Changes and revisions are carried out just as with Ada, except that you have some additional capabilities.

5.1 Making annotations

Go back to the X window prompt, and type

vcgen.X multiply.structure ⟨*return*⟩

to resume editing the function `multiply`. Click on the word '`WHERE`' to get your screen to look like Figure 43

Figure 43:

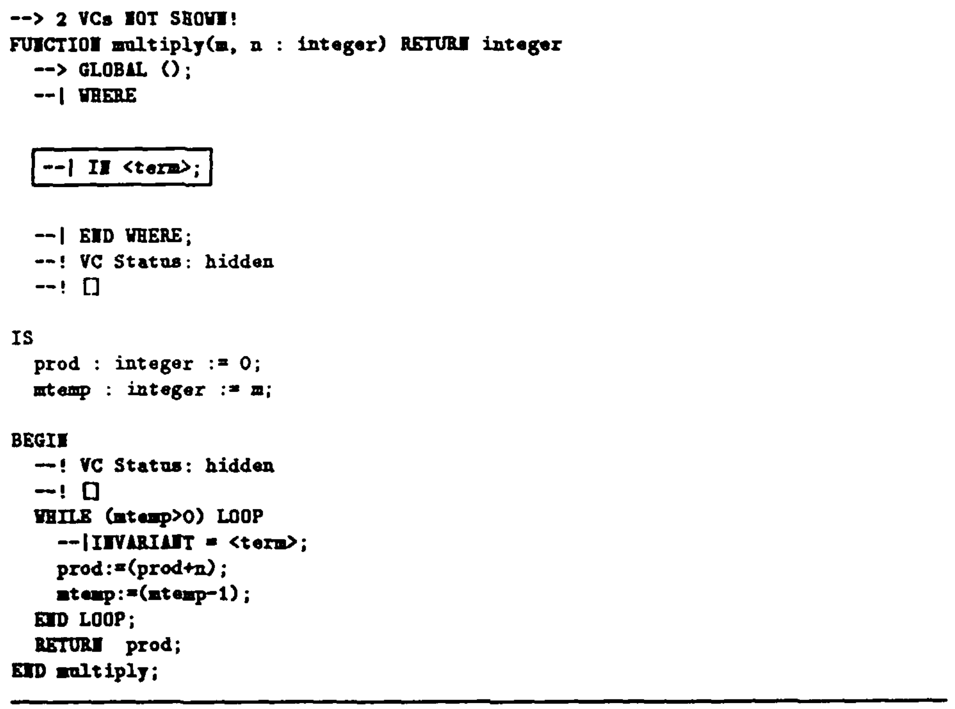
```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();

 ┌──────────────┐
 │ --| WHERE    │
 │ --| END WHERE;│
 └──────────────┘

  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP
    --|INVARIANT = <term>;
    prod:=(prod+n);
    mtemp:=(mtemp-1);
  END LOOP;
  RETURN  prod;
END multiply;
```

Select the menu item

in

to bring up the template for an IN annotation, as shown in the next figure.

Figure 44:



```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE

  --| IN <term>;

  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP
    --|INVARIANT = <term>;
    prod:=(prod+n);
    mtemp:=(mtemp-1);
  END LOOP;
  RETURN  prod;
END multiply;
```

Select the `term` placeholder and type (without ⟨*return*⟩)

`m>=0`

since the function computes the product correctly only under this condition. Select the word '`WHERE`' once more to get the menu for the Larch/Ada specifications.

Figure 45:

```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();

 +-----------------+
 | --| WHERE       |
 | --| IN (m>=0);  |
 | --| END WHERE;  |
 +-----------------+

  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP
    --|INVARIANT = <term>;
    prod:=(prod+n);
    mtemp:=(mtemp-1);
  END LOOP;
  RETURN  prod;
END multiply;
```

Select the menu item

`return`

to bring up the template for a RETURN annotation, as shown in the next figure.

Figure 46:

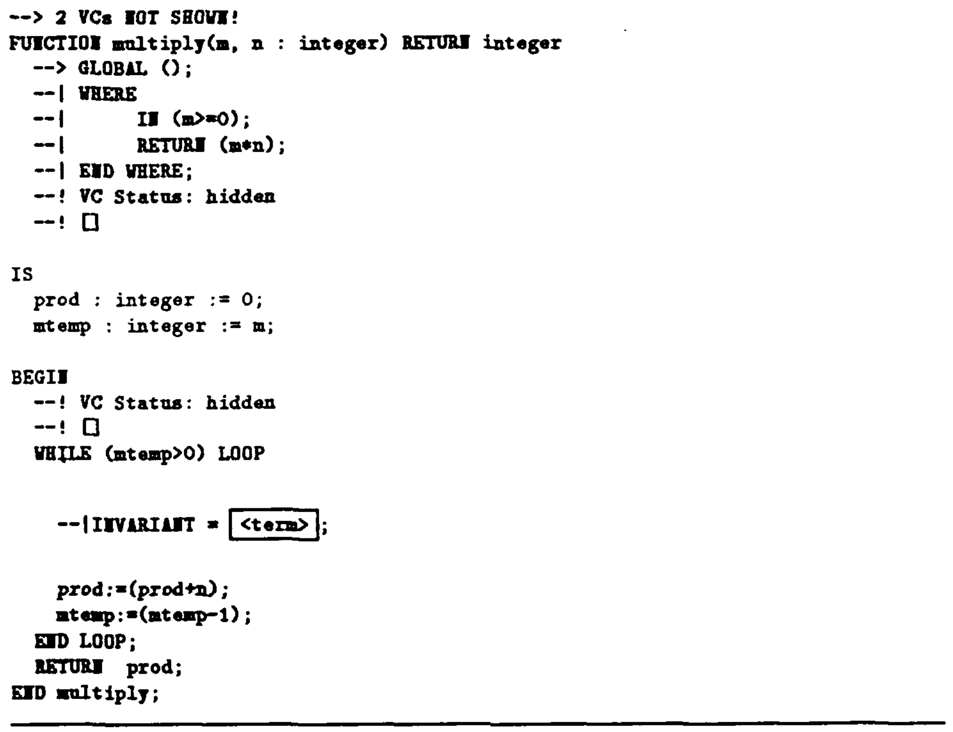
```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE
  --|       IN (m>=0);

  --| RETURN <term>;

  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP
    --|INVARIANT = <term>;
    prod:=(prod+n);
    mtemp:=(mtemp-1);
  END LOOP;
  RETURN  prod;
END multiply;
```

Select the `term` placeholder and type (without ⟨*return*⟩)

`m*n`

to specify the value that the function is to compute. Select the `term` placeholder in the `INVARIANT` annotation, and turn to the next figure.

Figure 47:



```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE
  --|      IN (m>=0);
  --|      RETURN (m*n);
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP

    --|INVARIANT = <term>;

    prod:=(prod+n);
    mtemp:=(mtemp-1);
  END LOOP;
  RETURN  prod;
END multiply;
```

You may now enter the loop invariant. Select the menu item

and

to bring up the template for a logical conjunction.

Figure 48:

```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE
  --|      IN (m>=0);
  --|      RETURN (m*n);
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP

    --|INVARIANT = (<term> AND <term>);

    prod:=(prod+n);
    mtemp:=(mtemp-1);
  END LOOP;
  RETURN  prod;
END multiply;
```

Enter

`mtemp>=0` in the first `term` placeholder. Select the second term buffer and enter

`prod + mtemp*n = m*n` ESC-`^N`

in the second placeholder. (You do not want to generate a statement placeholder.) The next figure is your completed, fully annotated Ada function `multiply`.

Figure 49:

```
--> 2 VCs NOT SHOWN!
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE
  --|      IN (m>=0);
  --|      RETURN (m*n);
  --| END WHERE;
  --! VC Status: hidden
  --! □

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: hidden
  --! □
  WHILE (mtemp>0) LOOP
    --|INVARIANT = ((mtemp>=0) AND ((prod+(mtemp*n))=(m*n)));

    prod:=(prod+n);

    mtemp:=(mtemp-1);
  END LOOP;
  RETURN  prod;
END multiply;
```

Save the contents to files **multiply.text** and **multiply.structure**, if you like.

I discuss verification conditions in Section 6. For detailed information, see the Penelope Tutorial [Hird 89].

5.2 More on annotations

Larch/Ada annotations can be entered and revised in the same way that Ada can be: you can select templates from the menu, you can enter and modify text in buffers, you can delete selections, you can copy to, cut to, or capture text in CLIPPED, you can paste, copy, or copy text from CLIPPED, you can insert before and insert after, etc. The only things that bear further mention are the menu items that appear when you have selected a Larch/Ada `term` buffer.

- `simplify`

  Use this to reduce your term to a simpler, logically equivalent form, if possible.

- (Developer's toolkit)

  Ignore these items. They are used in system development.

  - `deskolemize`
  - `conjunctify`
  - `deconjunctify`
  - `applyEquality`
  - `simplifyOps`

- `true`

  Use this to produce the text '`true`'.

- `false`

  Use this to produce the text '`false`'.

- `not`

  Use this to produce the template
  `(NOT <term>)`.

- `and`

  Use this to produce the template
  `(<term> AND <term>)`.

- `or`

  Use this to produce the template
  `(<term> OR <term>)`.

- `implies`

  Use this to produce the template
  `(<term> -> <term>)`.

- `forall`

  Use this to produce the template
  `FORALL <identifier>::<term>`.

- `exists`

  Use this to produce the template
  `EXISTS <identifier>::<term>`.

- `variable`

  Use this to produce the template

  `<identifier>`.

- `apply`

  Use this to produce the template for functional application,

  `<identifier>(<term>)`.

- `if`

  Use this to produce the template

  `(IF <term> THEN <term> ELSE <term>)`.

- `conjoin`

  Use this to add a conjunct to any selected term. For instance, when the selected buffer looks like

  `(x=17)`

  clicking on this item would produce the template

  `((x=17) AND <term>)`.

A subset of these items is available for sub-buffers of term buffers. A little experimentation will reveal the pattern.

6 Verification Conditions

Your selected buffer contains the first warning about VCs (see the last part of Section 5). A VC is a ***verification condition***, a theorem you must prove in verifying the Ada code. In the menu, you will see

show-vc

Selecting this causes the first hidden verification condition to be revealed.

Do the same for the other VC, and your file will contain the text displayed on the next page.

Figure 50:

```
FUNCTION multiply(m, n : integer) RETURN integer
  --> GLOBAL ();
  --| WHERE
  --|      IN (m>=0);
  --|      RETURN (m*n);
  --| END WHERE;
  --! VC Status: proved
  --! BY synthesis of TRUE

IS
  prod : integer := 0;
  mtemp : integer := m;

BEGIN
  --! VC Status: ** not proved **
  --! 1. (mtemp>=0)
  --! 2. ((prod+(mtemp*n))=(m*n))
  --! >>
    (IF (mtemp>0)
      THEN (((mtemp-1)>=0) AND (((prod+n)+((mtemp-1)*n))=(m*n)))
      ELSE (prod=(m*n)))
  --! <proof>
  WHILE (mtemp>0) LOOP
    --|INVARIANT = ((mtemp>=0) AND ((prod+(mtemp*n))=(m*n)));
    prod:=(prod+n);
    mtemp:=(mtemp-1);
  END LOOP;
  RETURN  prod;
END multiply;
```

In VCs, the the numbered formulae are your *hypotheses*, from which you must prove the formula following the double arrow ('>>'), the *conclusion*. When you have done so, the status lines will say '**proved**' instead of '**not proved**'. You will then have a verified function `multiply`, ready to save and compile.

The first VC is trivial: the editor automatically proved it. If you could see it before the automatic simplification and proof, it would look something like:

```
--!  1.  (m>=0)
--!  >> (m>=0)
```

The hypothesis is the `IN` condition of the function. The conclusion is the precondition of the function body.

Informally speaking, the second VC states that the loop invariant is preserved.

This gives the barest rudiments of VCs. For detailed information about proving VCs, see the Penelope Tutorial[Hird 89].

7 Acknowledgements

The software for Penelope is generated in part by the Synthesizer Generator under license from Cornell University. We acknowledge Thomas Reps and Tim Teitelbaum for their role in its development.

A Commands

A.1 Keystroke commands

Here, for quick reference, is a list of all the commands described in this guide, together with their official names, and a brief description. For a more complete list of commands, see Appendix C of the Synthesizer Generator Reference Manual [Reps 87].

- ⟨*backspace*⟩

  `backward-with-optionals`

  Select the previous buffer in the text (perhaps generating it as you go).

- ⟨*return*⟩

  `forward-with-optionals`

  Select the next buffer in the text (perhaps generating it as you go).

- `^A`

  `beginning-of-line`

  Move the cursor to the leftmost position of the selected text buffer.

- `^B`

  `left`

  Move the cursor one position to the left.

- `^C`

  `exit`

  Terminate the editor. You will sometimes be given a chance to reconsider, if you have not saved your changes. Do not rely on it.

- `^D`

  `delete-next-character`

  Delete the character to the right of the cursor.

- `^E`

  `end-of-line`

  Move the cursor to the rightmost position of the selected text buffer.

- `^F`

  `right`

  Move the cursor one position to the right.

- ^H

  backward-with-optionals

  Select the previous buffer in the text (perhaps generating it as you go).

- ^I

  execute-command

  See Subsection A.2.

- ^K

  delete-selection

  Remove the contents of the selected buffer. The buffer remains selected.

- ^M

  forward-with-optionals

  Select the next buffer in the text (perhaps generating it as you go).

- ^N

  forward-preorder

  Select the next displayed buffer in the text. This may be a sub-buffer of the current selection. No optional buffers are generated.

- ^P

  backward-preorder

  Select the previous displayed buffer in the text. This may be a sub-buffer of a larger previous buffer No optional buffers are generated.

- ^W

  cut-to-clipped

  Move the contents of the selected buffer to the CLIPPED buffer, deleting the selection.

- ^X^C

  exit

  Terminate the editor. You will sometimes be given a chance to reconsider, if you have not saved your changes. Do not rely on it.

- ^X^R

  read-file

  Read a file (to be named in the generated window) into the currently selected buffer.

- `^X^W`

  `write-named-file`

  Write the screen contents into a file (to be named in the generated window).

- `^Y`

  `paste-from-clipped`

  Move the contents of the CLIPPED buffer into the selected buffer, deleting the contents of CLIPPED.

- `DEL`

  `delete-previous-character`

  Delete the character to the left of the cursor.

- `ESC-DEL`

  `erase-to-beginning-of-line`

  Delete all the characters to the left of the cursor.

- `ESC-^N`

  `forward-sibling`

  Select the next displayed buffer in the text that is not a sub-buffer of the currently selected buffer. No optional buffer is ever generated.

- `ESC-^T`

  `copy-text-from-clipped`

  Copy the text captured in the CLIPPED buffer into the selected buffer (it must be a text buffer), saving the contents of CLIPPED for later use.

- `ESC-^W`

  `copy-to-clipped`

  Copy the contents of the selected buffer into the CLIPPED buffer, leaving the contents of the selected buffer as is.

- `ESC-^Y`

  `copy-from-clipped`

  Copy the contents of the CLIPPED buffer into the selected buffer, saving the contents of CLIPPED for later use. The contents of CLIPPED must be appropriate to the selected buffer.

- `ESC-d`

  `erase-to-end-of-line`

  Delete all the characters to the right of the cursor.

- `ESC-<`

  `beginning-of-file`

  Select the master buffer for the file.

- `ESC->`

  `end-of-file`

  Select the very last buffer in the text (no optional buffer is ever generated).

A.2 Command line commands

All the commands above may be typed on the command line, if you prefer. To do this, use the **`execute-command`** command: hit either the tab key or

`^I`

to get a command prompt on the command line. Then enter your command name, followed by ⟨*return*⟩. You do not need to type the whole command name, just an unambiguous prefix. For instance, you could enter

`write-n` ⟨*return*⟩

instead of

`write-named-file` ⟨*return*⟩

You may also answer the command prompt with any item currently in the menu, instead of clicking on it with the mouse. So for instance, in Figure 1 above, you could have typed

`^I`

and then

func-b ⟨*return*⟩

instead of clicking on the menu item '**`func-body`**'.

A.3 Mouse commands

Moving the mouse moves the mouse arrow on the screen. Once you move the arrow to your desired location, you have the use of the three buttons on top of the mouse for selecting buffer, menu items or transformation, and for issuing commands. If you like, you can use the mouse to do everything except enter text into buffers. Every command you have seen so far has its mouse counterpart.

A.3.1 Left button

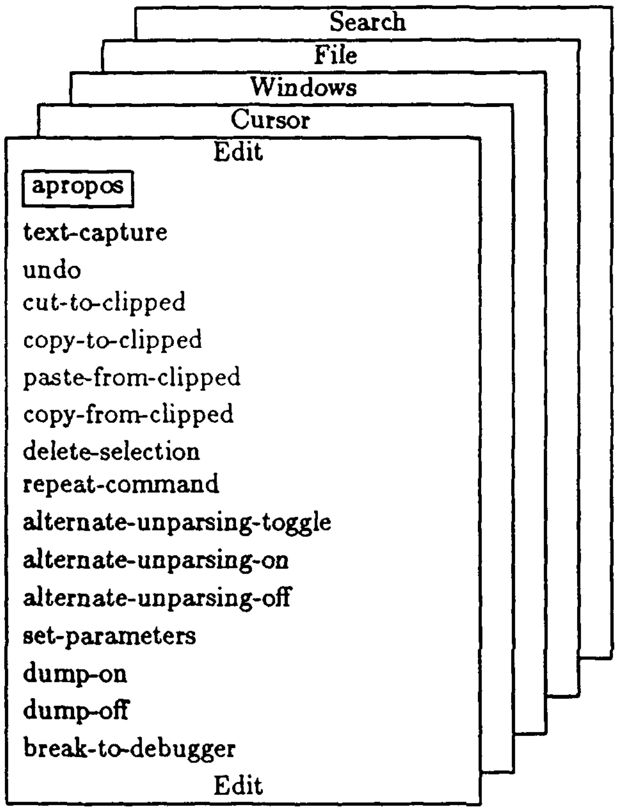
As you have seen, this button is used for selecting buffers and menu items, as well as for starting or aborting execution in command windows. In each case, you *click* the mouse; that is, you point with the mouse arrow, press the left button and then release it.

A.3.2 Middle button

You have seen that editor commands may be invoked with keystrokes such as `^K` or by typing the command name (in this case '`delete-selection`') on the command line. You may also use the middle mouse button to invoke commands. Which method you use is a matter of personal preference; the results of command execution are the same in any case. To do so, you *drag* the mouse; that is, you press the middle button, bringing the *drag menu* into view, move the mouse arrow to your desired selection, and then release the button. If you change your mind and do not want any selection, move the arrow off the menu entirely, and then release the button.

Let us look at specifics. Press and hold the middle button. Presently you will see the following multipage menu on your screen:

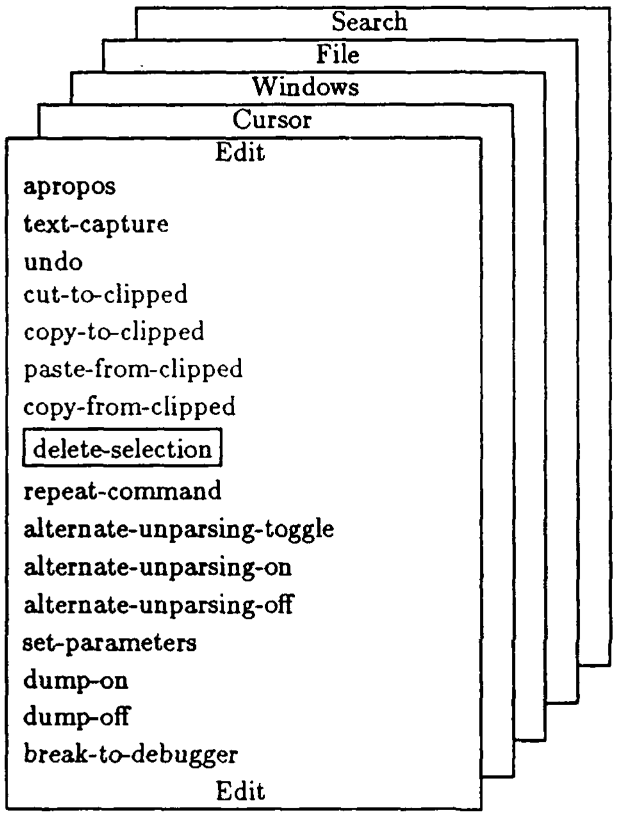
Figure 51:



I will discuss the other pages in a moment; for now, let us look at the top page. The menu item for the command **apropos** (which I have not discussed and will not), in reverse video, is the current selection. To invoke the command, you would merely release the button. More likely, you would want to move the arrow down, say putting '**`delete-selection`**' in reverse video, and then invoking by releasing the button. To avoid the nasty surprise of invoking the wrong command, keep the button down until you have the command name you want in reverse video.

Your screen will look like the following when you have moved the arrow to '**`delete-selection`**':

Figure 52:

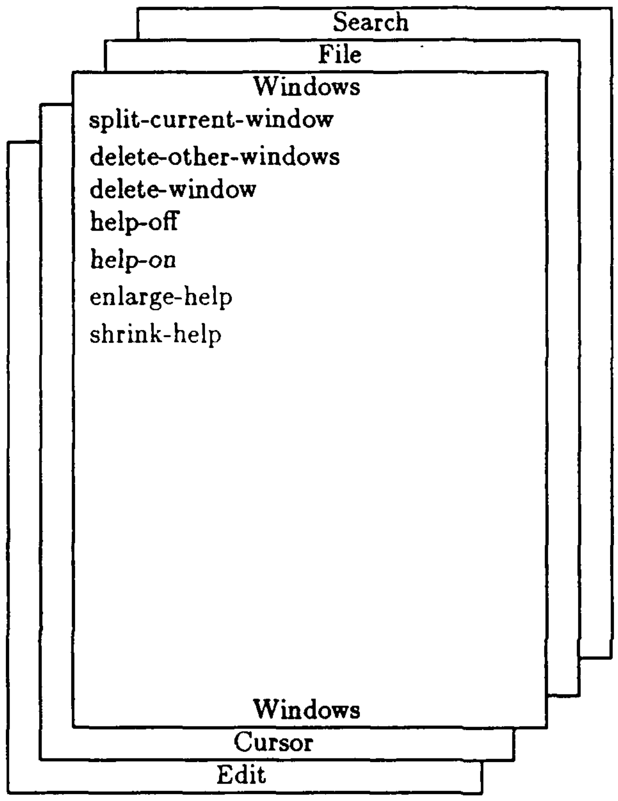


Each page of the multipage menu has a label, indicating what sort of commands are offered on it.

- The top page above is labeled '`Edit`' since the commands offered on that page have to do with text editing.
- The page labeled '`Cursor`' has to do with moving the cursor from buffer to buffer.
- The page labeled '`Windows`' has to do with controlling the various windows of the editor.
- The page labeled '`File`' has to do with contr›lling the input to and output from the editor.
- The page labeled '`Search`' has to do with searching (strangely enough).

To select a command from a page, you must first bring that page to the top of the sheaf of pages. Do this by moving the mouse arrow to the label for that page. For example, move the arrow to '`Windows`' to get the menu to look like the following:

Figure 53:



Notice that no command item is selected, which makes sense: you did not move the arrow to a command, but to the label. Notice also that the pages you skipped over are now in the sheaf at the bottom left. They can be called back by moving the arrow to the labels they have on the bottom.

To complete the discussion of the middle mouse button, here is an image of each menu page:

Figure 54:

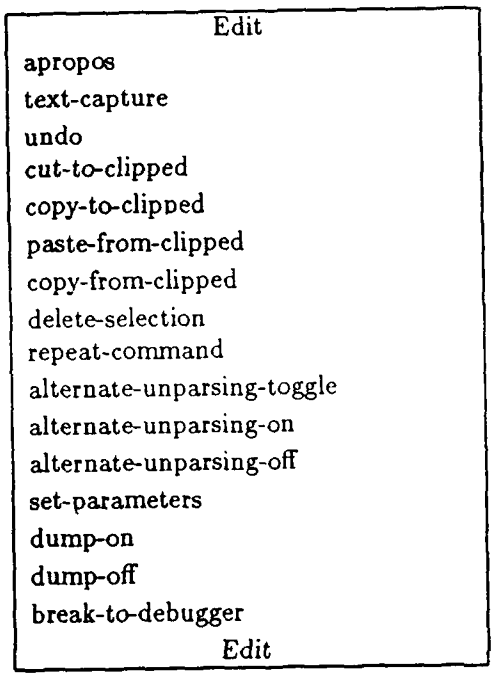

Edit
apropos
text-capture
undo
cut-to-clipped
copy-to-clipped
paste-from-clipped
copy-from-clipped
delete-selection
repeat-command
alternate-unparsing-toggle
alternate-unparsing-on
alternate-unparsing-off
set-parameters
dump-on
dump-off
break-to-debugger
Edit

Figure 55:

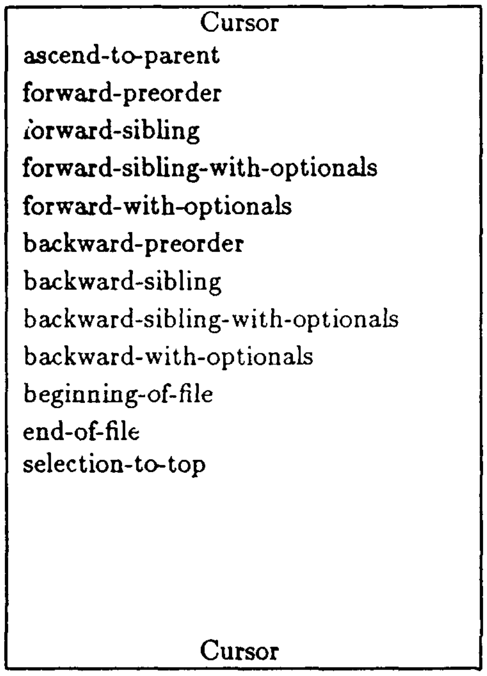

Cursor
ascend-to-parent
forward-preorder
forward-sibling
forward-sibling-with-optionals
forward-with-optionals
backward-preorder
backward-sibling
backward-sibling-with-optionals
backward-with-optionals
beginning-of-file
end-of-file
selection-to-top
Cursor

Figure 56:

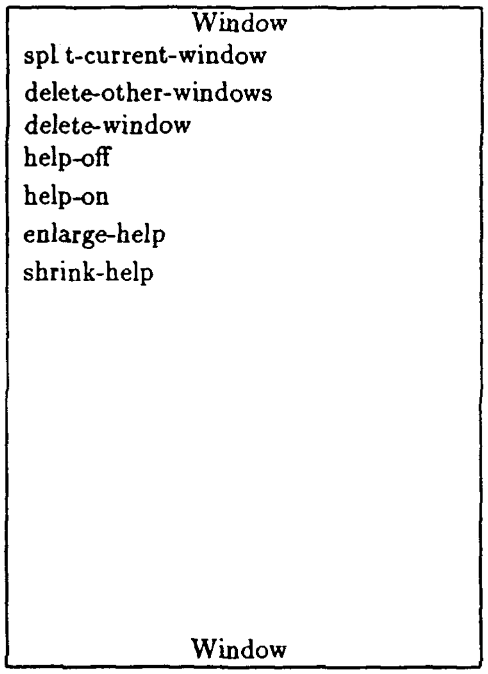

Window
spl t-current-window
delete-other-windows
delete-window
help-off
help-on
enlarge-help
shrink-help
Window

Figure 57:

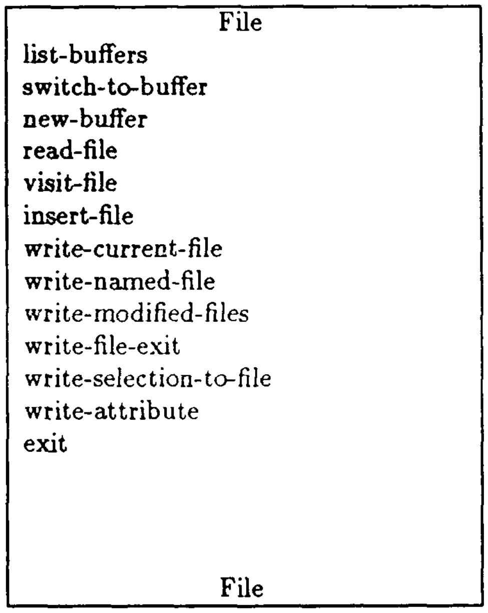
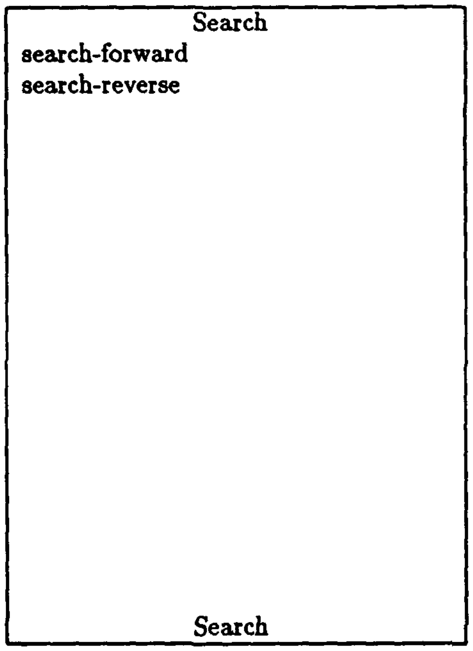

File
list-buffers
switch-to-buffer
new-buffer
read-file
visit-file
insert-file
write-current-file
write-named-file
write-modified-files
write-file-exit
write-selection-to-file
write-attribute
exit
File

Figure 58:

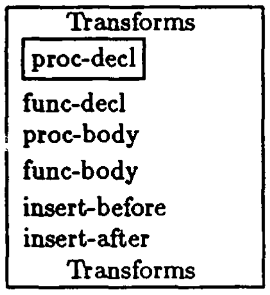


Many more commands are offered in the multipage menu than this guide treats. If you are curious about them, see the Synthesizer Generator Reference Manual [Reps 87].

A.3.3 *Right button*

You may use the right button to drag down a menu of transformations, which duplicates the menu at the bottom of the main edit window. For instance, pressing and holding the right button when your screen looks like Figure 1 brings up the menu:

Figure 59:



with, as you see, '`proc-decl`' selected.

- To invoke the transformation, release the button.
- To select another transformation, move the arrow to the item for it.
- To leave the menu without invoking a transformation, move the arrow away from the menu and then release the button.

Do not release the button until you have selected the transformation you want or have moved the arrow away from the menu; otherwise, you will inadvertently invoke whatever is selected.

A.4 Emacs cognates

Many of the SG commands correspond to equivalent emacs commands with the same key bindings. Here is a partial list.

Key binding	emacs name	SG name
^A	beginning-of-line	beginning-of-line
^B	backspace	left
^D	delete-character	delete-previous-character
^E	end-of-line	end-of-line
^K	kill-line	delete-selection
^P	previous-line	backward-preorder
^X^C	exit	exit
^X^W	write	write-named-file
ESC-<	beginning-of-file	beginning-of-file
ESC-<	end-of-file	end-of-file
ESC-W	wipe-out	copy-to-clipped
ESC-Y	yank-back	copy-from-clipped

B Glossary

chord

To press two or more keys at once.

click on

To place the mouse cursor on the target object, press and release the appropriate mouse button. If no button is indicated, the left button is assumed.

drag menu

A menu that is selected by pointing, and then holding down the appropriate mouse button (default is the left button). The menu items are displayed so long as the menu is selected. To choose an item, you point at it and then release the mouse button. To deselect the menu without choosing an item, point away and then release the button.

point

To place the mouse cursor on the target object.

References

[Ada 83] *The Ada Programming Language Reference Manual*, US DoD, US Government Printing Office, 1983, ANSI/MILSTD 1815A.

[Aho 72] A. Aho, J. Ullman, *The Theory of Parsing, Translation and Compilation, v. 1*, Prentice-Hall, 1972.

[Gries 81] D. Gries, *The Science of Programming*, Springer-Verlag, 1981.

[Gettys 86] J. Gettys, R. Scheifler, "The X Window System", Transactions on Graphics, to appear.

[Kleene 67] S. Kleene, *Mathematical Logic*, John Wiley and Sons, 1967.

[ORA 87a] *Draft Larch/Ada Reference: Version 0.1*, Odyssey Research Associates, May 1987.

[ORA 87b] *Revisions and Extensions to Larch/Ada: Phase II*, Odyssey Research Associates, November 1987.

[ORA 88] *Revisions and Extensions to Larch/Ada: Phase III*, Odyssey Research Associates, 1988.

[Hird 89] Geoffrey Hird, *Penelope Tutorial*, Odyssey Research Associates, 1989.

[Reps 87] T. Reps, T. Teitelbaum, *The Synthesizer Generator Reference Manual*, Department of Computer Science, Cornell University, 1987.

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C^3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.