Imperative Programming Part 3: Laboratory Manual

Peter Jeavons*
Trinity Term 2019

1 Introduction

The course *Imperative Programming Part 3* has a practical exercise which involves modifying a fairly substantial existing program. This is intended to give participants experience of the issues associated with a larger program that are presented in lectures. This manual contains general information about the set-up of the practical and instructions for the exercise.

In the practical exercise you will take the text editor that is one of the case studies in the course and enhance it with some additional features, ranging from simple navigation and editing commands to the ability to cut and paste arbitrary regions of text, all supported by the facility to undo and redo successive commands.

Section 4 of this manual gives a 'road map' of the *Ewoks* editor, explaining the function of each class. In Figure 1, you will find a handy 'reference card' for the editor, listing the keyboard commands and the functions that they perform.

Before you come to the lab to begin this practical, it will pay you to spend some time studying the code for the editor, and trying to understand how things fit together.

2 Setting up the practical

Materials for the labs are held in the file-store of the computers which you use in the demonstrated practical sessions. In order to copy the materials, you should use the command

```
$ cp -r /usr/local/practicals/ip3 .
```

This will create a directory called `ip3` in your file-store with a number of sub-directoriess.

What you do next depends on whether you wish to use Eclipse or an ordinary text editor for your programming[1]. Programming with libraries is made easier with the code completion features of an IDE like Eclipse. On other hand, some people find that projects that consist of more than just a Scala program are easier to manage with a text editor and traditional Makefiles. Try both and decide which you prefer!

*With acknowledgements to Joe Pitt-Francis and Mike Spivey.

[1] Scala also comes with the 'Scala Build Tool' *sbt* that provides an alternative to *make* and is reportedly faster at compiling Scala programs - you are welcome to explore using this tool on your own.

2.1 Starting Eclipse

If you choose Eclipse, launch it by typing eclipse at a UNIX command prompt, or by clicking the appropriate icon. It's OK to allow Eclipse to create a workspace in the default place. When you are past any welcome screen and have found the workbench, proceed as follows:

- Choose 'File/Import:...'
- Select 'Existing Projects into Workspace' and click 'Next>'
- Click on 'Browse' next to 'Select root directory', navigate to the `ip3` directory you just created, then click 'OK'.
- Select all of the projects that are now offered, but leave 'Copy projects into workspace' unchecked; click on 'Finish'.

You should now see the projects listed in the Project Explorer window on the left. They will still be there if you exit Eclipse and restart it.

You should now be able to compile and run the basic version of the editor by right-clicking on the *Editor* class in the Project Explorer (which is in the *default package* of the *src* directory of the *Ewoks* project) and choosing Run As... and then Scala Application. Try loading a file, changing it a bit, and then saving the result. To run the more advanced version, with undo capabilities, right-click on the *UndoableEditor* class and run that instead.

2.2 Using a text editor

If you choose to use a text editor together with *make*, you will find a suitable Makefile in each sub-directory of `ip3`. After editing the source, you should open a shell in the directory that contains the Makefile and type `make`. This will use the command-line Scala compiler *fsc* to recompile the whole program and put the class files in the same directory.[2] You can run the resulting program with a command like

```
$ scala -cp bin Editor
```

(The class name `Editor` indicates where the *main* method of the application is to be found.)

Because things are set up so that both Eclipse and the command-line compiler put their output in the same place, you can easily switch between using Eclipse and using the command line, compiling your program in Eclipse and running it from the command line, or vice versa.

3 The task: Extending *Ewoks*

The extensions described below involve adding new commands to the editor: some are easy comprehension exercises; some are more difficult, but fit well within the existing structure; and some require changes to the structure of the program, and are interesting as a way of judging whether that structure has been well chosen to allow for change.

[2] *fsc* is the Fast Scala Compiler, which starts a background server process so as to avoid reloading the Scala compiler each time you compile your program. It is functionally identical to the simple Scala compiler *scalac*.

Existing commands:

`LEFT, Ctrl-B`	Move one character left.
`RIGHT, Ctrl-F`	Move one character right.
`UP, Ctrl-P`	Move to the previous line.
`DOWN, Ctrl-N`	Move to the next line.
`HOME, Ctrl-A`	Move to the start of the line.
`END, Ctrl-E`	Move to the end of the line.
`PAGEUP`	Move up one screen.
`PAGEDOWN`	Move down one screen.
`Backspace`	Delete the character to the left.
`DELETE, Ctrl-D`	Delete the character to the right.
`Ctrl-G`	Abort the current command.
`Ctrl-L`	Redraw and recentre the display.
`Ctrl-Q`	Quit the editor.
`Ctrl-R`	Read in a different file.
`Ctrl-W`	Write the buffer to a file.
`Ctrl-Z`	Undo the most recent change.
`Ctrl-Y`	Redo the most recently undone change.

Possible additional commands:

`Ctrl-HOME`	Move to the start of the buffer.
`Ctrl-END`	Move to the end of the buffer.
`Ctrl-T`	Transpose two characters.
`Ctrl-M`	Place the mark at the current point.
`Ctrl-O`	Swap point and mark.
`Ctrl-X`	Cut the text between point and mark.
`Ctrl-C`	Copy the text between point and mark.
`Ctrl-V`	Paste the most recently cut or copied text.
`Ctrl-J`	Search for a string.
`Ctrl-S`	Interactive search for a string.

Figure 1: Reference card for *Ewoks*

3.1 The Required Extensions

1. Make the keystrokes `Ctrl-Home` and `Ctrl-End` move the cursor to the start and end of the editing buffer respectively.

   It is neatest to do this by extending the repertoire of 'directions' that is accepted by the method *Editor.moveCommand*. There are already values *Display.CTRLHOME* and *Display.CTRLEND* that correspond to the two keys.

2. Make the keystroke `Ctrl-T` transpose the characters to the left and right of the cursor, as it does in *Emacs*. Arrange for `Ctrl-Z` and `Ctrl-Y` to undo and redo the transposition.

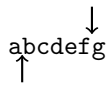
   This means adding a completely new command as a method in the *Editor* class (which is easiest to implement via a new method in the *EdBuffer* class). To implement Undo, you will need to override this method in the *UndoableEditor* class, and to create a new subclass of *UndoHistory.Change*, recording the operations needed to undo and redo this command.

   The *Emacs* behaviour is that the point moves one place to the right after transposing the characters[3], so that multiple transpose commands have the effect of moving the character that is to the left of the current point through the document.

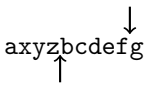
3. To make navigation easier (and to prepare for the *Cut* and *Copy* commands in the next task), introduce a second editing position called the *mark*. Provide a keyboard command `Ctrl-M` to place the mark where the point is currently located, and a command `Ctrl-O` to swap the point and mark.

   Note that whenever the text in the editor is modified the position of the mark may need to be updated. One way to deal with this is by enhancing the methods provided by *EdBuffer* to update the mark appropriately.

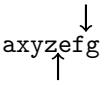
   For example, if ↑ denotes the point and ↓ denotes the mark, then the buffer might contain

   Inserting `xyz` at the point should give the state

   in which the mark still appears between `f` and `g`, though this means its numerical position has changed. More vital still, deleting three characters to the right of the point should now give us the state

[3] Actually, *Emacs* makes the command behave differently if the point is at the end of a line or the entire text; you may replicate this behaviour or not as you choose. Thanks to Ben Dawes for pointing this out.

where the mark *still* appears between `f` and `g`. Failing to update the mark in this case might leave it hanging in mid-air beyond the end of the text.

Also, the undo and redo commands ought to leave the mark in a sensible place. It's easiest to ensure this by treating the mark like the point and restoring it to the position it had before the command that is undone; this can be achieved by adding the mark position to the information saved in instances of *EdBuffer.Memento*.

4. Text editors usually provide some way of copying or cutting out part of the text, then perhaps pasting the copied or cut text in one or more other places. Design such a feature for *Ewoks* and implement it, providing a command `Ctrl-C` to copy the text between the point and the mark, a command `Ctrl-X` to cut that text, and a command `Ctrl-V` that pastes the most recent copied or cut text at the cursor.

3.2 Optional Further Extension

To complete the *optional* part of the practical implement one of the following additional extensions:

1. Add a simple search command, `Ctrl-J`, that uses the minibuffer to prompt for a string, searches the buffer for that string, and moves the editing position to the end of the next occurrence. The search should wrap around if it reaches the end of the buffer, and an error should be signalled (and the editing position should not move) if the string is not found, or if the only occurrence is where the point is originally. The string that is used in a search should become the default for future searches.

2. (An enhanced alternative to the preceding feature.) Add an interactive search command, `Ctrl-S`, that behaves similarly to the search in *Emacs*. As before, the minibuffer should be used to collect a search string, but now the cursor should stay in the main editing window, and show the next occurrence of that part of the search string which has been typed so far. (That way, the user only needs to type enough of the search string to identify the place in the buffer they want.)

   The interactive search feature in the *Emacs* has the following features: pressing the Backspace key during a search removes the last character typed, and also moves the cursor back to where it was before that character was typed; pressing Return finishes the search and leaves the cursor where it is; pressing `Ctrl-G` aborts the search and returns the cursor to where it was before the search. In addition, pressing `Ctrl-S` again during a search moves to the next occurrence of the search string without adding to it; the Backspace key reverses this action too. Pressing `Ctrl-S` at the start of the search pastes into the search string whatever string was used in the previous non-aborted search. Your implementation should provide similar features to make it as usable as possible.

   It's possible to reuse the undo mechanism implemented in the trait *UndoHistory* in order to provide the behaviour associated with Backspace. (If you do so, you can ignore the support for redo.)

4 A road map

This section contains a brief description of the various named classes that make up the *Ewoks* editor. The classes are listed in roughly bottom-up order, so that the description of each class can refer to ideas introduced in describing lower-level classes.

Text: An instance of the *Text* class represents a sequence of characters, with efficient operations for inserting and deleting characters at arbitrary places in the sequence. It is implemented using a gap buffer, so that groups of insertions and deletions that are spatially close to each other can be carried out cheaply.

There is no in-built limit on the length of the text, because if the buffer becomes full it expands in size. Various additional methods are provided, some for convenience and some for efficiency.

PlaneText: The *PlaneText* class is a subclass of *Text* that represents the same sequences of characters, but it keeps an additional data structure (a line map) to support methods that refer to the two-dimensional layout of the text.

The line map is stored in the simplest way possible: an array of line lengths. In order to speed up clustered accesses to the text, the module keeps track of the current line and its position, and it finds other lines by moving forwards or backwards from the current line. If nothing else, this means that a pair of calls to *getRow* and *getCol* for the same editing position require at most one search for the correct line.

When the text changes, there is a half-hearted attempt to optimise updating of the line map: if an insertion or deletion is entirely within one line, then just that entry in the map is updated. Otherwise, the whole map is recalculated from scratch[4].

EdBuffer: An instance of *EdBuffer* represents the state of an editing session: a text buffer together with (at least) an editing position, plus other state components that might be added later. This class is the 'model' that is viewed via the Display class, and is the subject for the controller in the *Editor* class.

The *EdBuffer* class helps with the implementation of undo by providing a method *getState()* that returns a *Memento* object, capturing all of the undoable state except for the contents of the text. Each *Memento* object provides a method *restore()* that can be used to reset the *EdBuffer* to the state in which it was created.

Terminal: *Ewoks* is designed as if it used an old-fashioned VDU, or the simulation of a VDU that is provided as the 'command window' of modern windowing systems. In order to make sure it works the same way on all platforms, this tutorial version of *Ewoks* has its own simulation of a VDU, implemented on top of the Swing GUI toolkit that is part of the standard Java libraries. The implementation is not very clever, so inevitably most of the CPU time used by the editor will go into refreshing the display from the matrix of characters.

We shall not study the implementation of the *Terminal* class in the course, and a couple of complicating details are hidden there: one of them is a queue that mediates between receiving keystrokes, which happens in one thread of the program, and consuming them in the editor, which happens in a different thread. Among other things, the editor

[4] It would be interesting to measure the effectiveness of this simple strategy, and find out if more sophisticated implementations save any time. One thing is certain: most accesses to the line map and to the underlying text are made for the purpose of updating the display, and these accesses are spatially clustered and read-only.

must be able to suspend itself if there are no keystrokes waiting, and be woken only when the user presses another key.

It would be straightforward but tedious to re-implement the *Terminal* class so that, instead of working on a simulated screen, it output the control sequences that have the same effect on a real (or externally simulated) terminal.

Display: This class is responsible for keeping the display up to date. It is attached to the *EdBuffer* that holds the text being edited and to the *Terminal* on which the display appears. At the end of each editing command, the editor calls the method *Display.refresh(flag, row, col)*, with *flag* indicating the extent to which the display may be out of date: *CLEAN* if the text has not changed, *REWRITE LINE* if the changes are limited to one line, and *REWRITE* otherwise. The display then retrieves the relevant lines from the editing buffer and shows them on the terminal.

There is an instance variable *origin* that identifies the top line on the display, and the class had the responsility of maintining this variable in accordance with the *scrolling policy* of the editor. This policy is as follows:

- The editing position is always on the screen.
- The origin does not move except when necessary to keep the cursor on the screen, or in response to an explicit command.
- If the origin moves spontaneously, then it is chosen so as to put the editing position on the middle line of the screen.

Editor: This is the controller that provides methods that correspond to editor commands and implements them by modifying the model of editor state, an instance of *EdBuffer*. The *Editor* class also contains the main command loop, which waits for a keypress, then looks it up in a keymap and invokes the corresponding command.

Keymap[Command]: A keymap is a mapping from keystrokes to commands: the main loop of the editor reads a keystroke from the editor, looks it up in a keymap, performs the resulting command, and then repeats.

In fact, the editor has several command loops, corresponding to interaction modes where the same keystrokes do different things. Pressing `RETURN`, for example, inserts a newline character in the normal mode of the editor, but when the editor is prompting for a filename, the same key causes the file to be saved with the name that appears in the minibuffer. And when an interactive search is in progress, the same key has yet another action, to finish the search and return to normal editing. Having lots of different modes is supposed to be a bad thing in a user interface, but if you do have several modes, it's important that the user can see from the display which mode the editor is in.

A keymap binds keys to values of its parameter type Command; in actual keymaps, this is typically a function defined on some *Target*, that represent an action that can be carried out on that target (usually, one of its methods).

UndoHistory: This is a trait that can be mixed in to classes that provide a history of undoable changes. Commands that can be undone should create an instance of *UndoHistory.Change* that describes the change they made, and how it can be undone and redone. UndoHistory is mixed in to the controller class *UndoableEditor*.

UndoableEditor: This class is a subclass of *Editor* that mixes in the *UndoHistory* trait in order to implement undo; each editor command that modifies the buffer creates an instance of

UndoHistory.Change that captures the change to the text, and part of the protocol for invoking commands, expressed in the method *obey(cmd)* wraps these changes with *EdBuffer.Memento* objects that capture the rest of the undoable state.

MiniBuffer: Some editing commands take a string or filename as a parameter, and they use the bottom line of the display as a separate area where the user can edit a short text. An instance of the *MiniBuffer* class holds the state of one of these small editing sessions. In a sense, it is an entire text editor in itself, with a keymap that binds keys to commands, and a command loop that reads keystrokes from the keyboard and carries out the corresponding editing actions.

When the main editor needs to read a string, it can create an instance of *MiniBuffer* and post it on the display. It can either call the command loop of *MiniBuffer*, or (as is needed for the interactive search feature that is an optional exercise) use another keymap and command loop. When the small editing session is over, the minibuffer is unposted from the display and thrown away, and the command loop of the main editor takes over again.

Testbed: For testing, the *Testbed* class provides an alternative main program that, instead of getting input from the keyboard, replays keystrokes that have been recorded in a file. The lab materials contain a shell script *runtest* that allows editing sessions to be recorded and replayed using this class.

5 Tools for testing

When you are testing a large program, it helps immensely to have a framework that lets you store tests, run them automatically, and check automatically that the results are correct. For *Ewoks*, I have provided some regression tests that exercise the whole editor. The regression tests are provided by a UNIX shell script *runtest* that is located in the root directory of the program, together with several files in the *test* subdirectory.[5]

Each of the files in the *test* directory packages together a file to be loaded into the editor, the sequence of keystrokes for an editing session, and the final contents of the file and of the editing screen. To create one of these packaged tests, you need to invoke the *runtest* script with a -c argument:

```
$ sh runtest -c foo test/mytest
```

In this command, `foo` is the name of a file that will be loaded into the editor, and test/mytest is the file that will contain the packaged test. The first thing that happens is that the *Ewoks* window opens for you to type some editing commands. When you have finished, save the result to the default file (using Ctrl-W) and quit the editor. During this editing session, a special feature of the *Terminal* class is used to record all the keystrokes in a temporary file.

The next thing that happens is that the file of recorded keystrokes is automatically replayed in another editing session to check it. Instead of showing the screen contents in a window, this special session — managed by the Testbed class — keeps track of the screen and records its final state in a file. At the end of the replay, you will see a listing, generated using the UNIX program *diff*, of the changes that have been made to the edited file. Finally, you will be prompted for a one-line description of the test that is stored as part of the package:

[5] If you work on the practical using your own Windows machine, then these UNIX scripts will be no use to you and you will need to make alternative arrangements to run suitable tests.

```
Replaying the session ... OK
1c1
< A small file
---
> xA small file
Please type a brief description of the test:
Add an x at the start
$
```

The *runtest* script takes a pre-recorded test and runs it again, using the *Testbed* class as before. So giving the command

```
$ sh runtest test/mytest
```

unpacks the test we just made and replays it on the editor — or on a subsequent version of it. First the one-line comment is printed, then the editor runs on the same file and recorded sequence of keystrokes as before. The script prints any differences it finds between the expected file and screen contents and those it actually finds.

```
Add an x at the start
```

In this case, no differences were found, so nothing was printed. You can run all the tests in files named `test/test*` with the command `make regress`.

For testing your changes, and especially if you want to measure performance, it will be useful for you to prepare some suitable large text files. For example, a 1MB file consisting of 20,000 lines of 50 `a`'s, but with one `a` replaced by a `b` somewhere near the middle. That file would be useful for testing the search function of the editor. Some other tests of basic performance might use a 10MB file consisting entirely of lines of `a`'s. Of course, one wouldn't want to type such a file by hand, but with a bit of ingenuity, it's possible to put it together quickly using an ordinary text editor or standard UNIX commands.