A screen editor written in the Miranda[1] functional programming language

Simon P. Booth & Simon B. Jones
Department of Computing Science & Mathematics
University of Stirling
FK9 4LA

29 September 1994

Abstract

This paper explores the development of an interactive screen editor, in the functional programming language Miranda, from a specification written in Z. The program makes use of a set of interactive functions especially developed to aid the writing of interactive programs (in Miranda). The editor developed is fully featured and could be used to edit text, although the execution speed is rather poor.

[1] Miranda is trademark of Research Software Ltd.

1. Introduction

In this paper we examine the use of the lazy functional programming language Miranda to develop an interactive screen editor. The basic specification of the editor is taken from the Z specification given in Sufrin[1] and the programming methodology is based on work by Thompson[2]. A Z specification was chosen because it gives a concise mathematical meaning to the specification; thus, proofs can be derived for certain features. Thompson's combinators are derived from examining some of the problems associated with interactive functional programs and provide many useful functions to write interactive programs.

The Z specification will be implemented in a pragmatic fashion: this will not be a strict exercise in the implementation of a specification – features that make sense in the specification but not necessarily when implementing will be dropped. Section 2 discusses the differences between the implementation and specification in detail.

The rationale behind Thompson's combinators and some of the problems associated with interactive functional programming are discussed in section 3. The combinators have been slightly modified. Details of the modification are given in section 4.

An editor is an archetype of an interactive program where the user types a command at the keyboard and then sees the result of this action on the screen (e.g. word is deleted or a newline inserted). Indeed most modern computer users would perceive this type of program as "normal"; the word processor being used to type this paper does precisely this. So if functional programming is to become more than an interesting area of research the development of this type of program is important – even though functional languages may not at first sight seem well suited to this type of problem. Miranda (among others) does address the problem by providing facilities for terminal and file input and output. These facilities do provide a mechanism

for the development of interactive programs. The implementation of an editor is discussed in section 5. (Note: input facilities are not consistent with purely functional semantics – there is no guarantee that a file read at time t_1 equals the file read at t_2, as required for referential transparency).

The editor developed is intended as a usable tool that could actually be used to edit files (albeit slowly), and so a full range of editing functions has to be provided: arrow key motion; deletion of various objects (characters, words); cut & paste; save & retrieve files, etc. Also, the user interface should be simple to use with quicker access to the functionality for the experienced user (of whom there is currently only one of the authors!). All of these features are available and section 6 can be regarded as a "user manual".

Listings of the programs are available from the authors.

2. The Z specification & its implementation in Miranda

2.1. Introduction

Z is a mathematical specification language based on set theory[3,4]. The specification is usually given as "schema"s followed by an informal English language description. This is best illustrated with a simple example from Hayes[5] that specifies an operation to look up an identifier in a symbol table:

```
┌── Lookup ──────────────────────────────
│ Ξ symtab
│ s? : SYM
│ v! : VAL
├────────
│ s? ∈ dom(symtab) ∧
│ v! = symtab(s?)
└────────────────────────────────────────
```

> "Get the value associated with identifer s. Ensuring that s is in the symbol table"

Ξsymtab declares symtab and symtab' (the state before and after the operation) and specifies that symtab = symtab' (the operation does not change the state). symtab is a schema itself which gives symtab: SYM ↦ VAL – a partial function from symbols (SYM) to values (VAL). s? is the input symbol (? means input; ! means output), and v! the output value. All this is contained in the first box in the schema – the declaration part. The second box (predicate part) describes the operation: s? ∈ dom(symtab) specifies that s? should be in the symbol table (the domain of the state, symtab). The second, v! = symtab(s?), states that the output value is associated with s?. These two lines are "anded" together to form the predicate.

Z is useful because it allows specifications to be formulated in a mathematical form and avoids the problems associated with natural language descriptions. In Z the natural language part is only used to give a flavour of what the mathematics is describing.

2.2. The Specification: A Summary

The specification is given in three parts. Firstly the editing subsystem is defined; next the document display subsystem; finally the relationship between the two is specified.

The editing subsystem is given as transformations on documents (i.e. what changes in the document when, say, the left arrow key is pressed or a letter inserted).

The display subsystem details how to display documents on a screen. It defines a window (what the user actually sees) and how to map a document to this window. It also specifies that the current cursor position must always be visible to the user.

2.3. Differences from the Specification

In implementing the editor, Sufrin's specification has not been followed to the letter: for instance, a mirror function is defined:

```
mirror : DOC → DOC
─────────────
mirror ( l, r) = (reverse(r), reverse(l))
```

where DOC is defined as *seq[CH]* x *seq[CH]*. Mirror is defined so that all operations (like move, del, etc) can be defined as leftward operations and the corresponding rightward operations are defined using mirror:

```
right, left : (DOC ↦ DOC) → (DOC ↦ DOC)
─────────────
right (f) = (mirror ∘ f ∘ mirror)
left (f) = f
```

So given a del operation that deletes the last character in the first *seq[CH]* in a DOC (a leftward deletion), then the corresponding rightward operation is given by:

$$\text{mirror} \circ \text{del} \circ \text{mirror}$$

applied to the document. This will delete the first character in the second *seq[CH]* in a DOC.

Reversing the whole file twice (which the above implies) is too inefficient for use in the editor and amounts to an enormous amount of work to delete a character! It does, of course, simplify the specification as left/right directional actions only need to be specified in one direction and then mirror used to define the action in the opposite direction.

In order to avoid the use of mirror, our implementation has to specify actions in both directions. So instead of a single move which is used in conjunction with left and right (above) to achieve "cursor one character left/right", we have:

$$\text{move\_left, move\_right: DOC} \mapsto \text{DOC}$$

$$\text{move\_left } (l, r) = (\text{head}(l), \langle \text{tail}(l) \rangle \frown r \mid l \neq \langle\rangle)$$
$$\text{move\_right } (l, r) = (l \frown \langle \text{front}(r) \rangle, \text{last } (r) \mid r \neq \langle\rangle)$$

All the operations defined in the specification (move, del, find, replace, cut & paste, etc) have been implemented in the editor and operations have been added: move up, move down a line (using the appropriate arrow keys) and move to line n.

The implementation of the keyboard mappings appears different but is fundamentally the same. The specification suggests that certain editor commands (QUOTE, FIND, REPLACE, etc) are directly mapped to keys (i.e. there is a FIND "key"). The implementation does not have a FIND key but

rather a FIND command which is a sequence of keys: ESC sd␣abc (i.e. Escape followed by sd (search down), space, then the string to find, abc). The specification has a direct mapping between physical and logical keys whereas the implementation does not always support this. All this implies is a change to how commands are actually given to the editor. For instance, given above is the sequence of keys that are pressed to make the editor find abc. In the specification the sequence would be QUOTEabcFIND. (QUOTE is equivalent to ESC and is used to introduce text that is not to be added to the document.)

QUOTEd text (as the specification calls it, e.g. abc above) is placed in a DOC and the basic editor commands (move, del, left, right, etc) are available to edit the string. The implementation only supports backspacing to delete erroneously typed characters (arrow keys are ignored). Also, the editor does not follow the specification's practice of placing the quoted text at the current position on the screen but places the cursor on the status line (the bottom line on the screen). Also note how in the specification the text is followed by the command. Although this at first might seem unnatural there is a benefit in that delimiting commands and their text is unnecessary. Our command text (sd abc) style requires a delimiter but does have a more “natural” command first approach.

Operations to find and replace throughout the remainder of the document are also available (with two modes of operation: noconfirm and confirm – commands fp and fq). These operations can only be achieved via multiple FIND & REPLACE key depressions in the specification, although the amount of typing is reduced by the fact that both FIND and REPLACE recall their previous values. FIND and REPLACE are available as ^F and ^R (see section 6.3)

The display policy defined has been followed: the policy specified is that the cursor must remain on screen at all times but beyond this the specification takes no view. Exactly what policy is selected to deal with situations when the cursor is about to move off the screen is left to the programmer. (We implement a policy of moving half a screen height/width

in the appropriate direction when about to move over screen boundaries. This helps to minimise screen refreshes as the document is traversed.)

The specification also develops a state for the editor. Initially this is given as *seq[CH]* x *seq[CH]* (the strings to the left and right of the cursor position) but, as the specification is developed, this is enriched to capture the various new features being added to the editor. The actual state structure used here is different from the specification. The differences arise due to the fact that for, mathematical convenience in the specification, elements of the state such as the last deleted text are given in the form of a document, i.e. *seq[CH]* x *seq[CH]* – where one of the strings will always be empty. These have been programmed as *seq[CH]*. Also, the state used has an additional feature: a number of boolean values are contained within the state. These values are used to keep track of certain events: such as whether the file has been saved since the last modification. Two additional *seq [CH]* are also present: one to indicate whether the editor is in insert or overwrite mode (it is written to the status line on the screen to indicate the mode) and the other to hold the file name. The insert/overwrite modes are another extension to the specification, which only supports insert. A stack structure is present *seq [edit_mode]* which is used to keep track of what "mode" (command, insert, etc) the editor is currently in. The editor supports several different modes and a stack is the simplest way of keeping track of what mode in editor was in as certain operations terminate.

A full definition of the operation of the editor is given in section 6.

3. The Thompson Combinators

3.1. Functional Programming and Lazy Evaluation

Functional programming is based on the definition of functions that evaluate lists. A classic example is a function to sort a list xs (using the quicksort algorithm):

```
sort []         = []
sort (x:xs)     = sort [y | y <- xs; y <= x] ++ [x] ++ sort [y | y <- xs; y > x]
```

Some of the elegance of functional programming can be gained from the above definition of sort: it could be regarded as a specification.

Lazy functional languages only actually evaluate arguments when they are needed. Thus if we ask for the first item in a list (hd), a lazy functional language will then only evaluate the first item when it is needed. In a lazy language we are quite at liberty to specify the first item from an infinite list as the language having evaluated the first item does not evaluate the remainder:

```
hd [1..]        = 1
```

This differs considerably from an eager language which evaluates the whole list first. This "evaluate when needed approach" can be exploited to provide mechanisms to read standard input. In Miranda, *$-*, stands for the contents of standard input (as a list of characters)

```
hd $-
```

will evaluate (or get) the first character from standard input. Note that there are problems when standard input is the keyboard as most operating systems will buffer the input until the return key is pressed and so the first character cannot be obtained until return is pressed:

```
Miranda hd $-
1234↵
'1'
Miranda
```

This represents a problem for any type of interactive program that requires to deal with each character as it is typed. This will be discussed later in section 5.3.

3.2. Interactions

In Miranda an interactive program reads a stream of input and writes a stream of output. Both the input stream and the output stream are treated as infinite lists and so if we define the types:

input == [char]
output == [char]

then a function with the type

input → output

would have an appropriate type for simple interactions, i.e. take some input and produce some output (like the function above to produce the first character). However, we need a more sophisticated type of behaviour for proper interactive programs; an interaction should return the unconsumed portion of the input (i.e. the characters that it has not used) for use in later interactions. This type of interaction would be of the type:

input → (input, output)

A function to echo the first input character could be rewritten as (assuming that the input stream is not empty):

my_head (x:xs) = (xs, [x])

Thompson[2] introduces a further (and powerful) refinement to this model: a state which is passed from one interaction to the next. This is modelled, in general, by supplying the initial interaction with a state, of type *, and the interaction returning a final/new state, possibly of a different type, **, on termination. This gives a general type of partial interactions:

*interact * ** == (input, *) → (input, **, output)*

The addition of the state information allows an interactive function to act on the initial state and return (possibly) a new state plus any output that the operation may generate. A simple example of this type of program would be a function to get the next character in the input stream, to add that character to the state (returning the new state) and, say, producing no output:

```
get_char :: interact [char] [char]
get_char (in, st) = (tl in, (hd in):st, [])
```

(*tl* returns a list without its head element).

A suite of input/output and control *combinators* (*write, writeln, alt*, etc) with types similar to that above are provided (by Thompson) with Miranda.

The actual interact type is:

*interact * ** == ([[char]], *) → ([[char]], **, [[char]])*

This differs from the above in that this breaks the input/output streams into lines rather than characters. The *write* combinator is

```
write :: [char] -> interact * *
write outstring (in, st) = (in, st, [outstring])
```

and *get_char*

```
get_char :: interact [char] [char]
get_char ((i:in), st) = ((tl i):in, (hd i):st, [])
```

an important control combinator is the one that sequences interactive operations correctly (*sq*):

```
sq :: interact * ** -> interact ** *** -> interact * ***
sq inter1 inter2 x =
        make_output out1 (inter2 (rest, st))
        where (rest, st, out1) = inter1 x
```

make_output is a function that places its first argument at the front of the output stream.

4. Modification to Thompson's Combinators

A fundamental feature in any editor is the ability to save files! This means that the interact type must be changed as files cannot be saved if the output type is *[[char]]*. In Miranda the value of an expression is a list of "system messages". The type *sys_message* has definition (where *file == [char]*):

```
sys_message ==      Stdout [char] | Stderr [char] | ToFile file [char] |
                    Closefile file | Appendfile file | System [char] |
                    Exit num
```

Miranda silently converts anything that is not *sys_message* to this form (so most of the time we do not need to know about *sys_message*). Example:

```
Miranda 2+5
7
```

The 7 is the result of the conversion: [*Stdout (show(2+5))*].

To allow the usage of the *ToFile* component we must change the output type to *sys_message*. The interact type becomes:

```
interact * ** == ([[char]], *) → ([[char]], **,  [sys_message])
```

All the Thompson functions have been changed (where necessary) to reflect this. Thus *write* becomes:

```
write :: [char] -> interact * *
write outstring (in,st)
      = (in,st,[Stdout outstring])
```

We also added the combinators *get_char* and *get_char_out*. The first is described in Thompson[2] but is missing from the actual scripts provided with Miranda. The second is new: it is like get_char, but places the character on the output stream. Its definition is:

```
get_char_out :: interact [char] [char]
get_char_out ((i:in), st) =
        ((tl i):in, first:st, [Stdout [first]])
        where first = hd i
```

5. Programming the editor

We now discuss some of the programming issues. Firstly a full definition of the state is given.

5.1. The State

The state is fundamental to the use of the combinators and is defined in the Z specification. Below is the Miranda for the state:

```
file_name ==        [char]
status_line ==      [char]
left_text_str ==    [char]
right_text_str ==   [char]
deleted ==          [char]
last_find ==        [char]
last_rep ==         [char]
re_cycle ==         (deleted, last_find, last_rep)
command ==          ([char], char)

edit_mode ::= Com | Ins | Ovr | End | YesNo | Number

state ==    ((left_text_str[2], right_text_str, status_line, re_cycle, file_name),
             (num, num, num, num, num, num, num),
             (bool, bool, bool, bool),
             [edit_mode], command)
```

The *nums* in the second component of the state contain the following: current on-screen character position (x), current on-screen line position (y), number of characters currently off screen to left (x offset), number of lines currently above screen (y offset), height of screen, width of screen, length of left text – used by cut and paste operation (mark sets this).

The *booleans* in the third component of the state are: *True* if there are no outstanding changes to the text since the last save operation, *True* if the

[2] The left string is stored in the state in reverse order. This is because it is computationally cheaper to determine the first n elements in the string than the last n. Of course, it does mean that reverse has to be used when moving n elements from either the left to the right or the right to the left.

screen requires redrawing, *True* if a mark (for a cut/copy and paste) has been set. The last one is currently unused.

The *edit_mode* is used as a stack containing the edit modes:

- ***Com*** – In command mode. All text entered is command (appears on status line);
- ***Ins*** – Insert mode. All characters appear on screen and are added to left string;
- ***Ovr*** – Overwrite. All characters appear on screen. First character of right string is removed and typed character is added to left string;
- ***End*** – Exit editor;
- ***YesNo*** – Only Y or N can be entered;
- ***Number*** – Only a number can be entered;

The *command* part of the state has two components *[char]* for the actual text of the command and *char* for the command delimiter.

5.2. Keyboard Management

Although any screen editor must deal with any character typed at the keyboard, some keys generate several characters. For instance, → generates Esc, [, C. A special function *get_keyboard* deals with this:

```
get_keyboard :: interact [char] edit_action
get_keyboard = sq get_char (alt normal_letter aletter special)
```

sq and *alt*[3] are two combinators for building interactions. *sq* is defined at the end of section 3.2 (page 11)

---

[3]

```
alt :: condition * -> interact * ** -> interact * ** -> interact * **
alt cond inter1 inter2 x
        = inter1 x , if cond x
        = inter2 x , otherwise
```

get_keyboard reads one character. If it is a special character then further characters may be read. The two interactions *aletter* and *special* encode the character or character sequence read as an *edit_action* and return this as the final state. The type *edit_action* is defined as follows (a full defintion is given in section 5.3):

```
edit_action ::= Again search_dir |
        Ask text |
        Cerror text |
        Change char |
        …
```

For example, *get_keyboard* maps "→" (i.e. Esc[C) to *Do (Move_Right 1)* and similarly maps all the special keys[4] to *Do key_action*. The type for *key_action* is:

```
key_action ::=  Move_Right num | Move_Left num |
                Move_Up | Move_Down |
                Delete_Char | Delete_Line |
                Insert_Char | Insert_Line |
                Backspace |
                Escape char |
                Previous | Next |
                Home | Shift_Home |
                F1 num | F2 num | F3 num | F4 num |
                F5 num | F6 num | F7 num | F8 num |
                Select |
                Unknown
```

A normal qwerty key is mapped to *Normal char*.

Thus *get_keyboard* returns to the editor a logical encoding of the key pressed and not the sequence of characters that the key generates. (Note: typing Esc[C explicitly causes *get_keyboard* to return *Do (Move_Right 1)*, just as if → had been pressed)

[4] Keys that generate more than a sequence of characters and the sequence always starts with Esc.

get_keyboard can only map the arrow keys to *Do (Move_Right 1), Do (Move_Left 1), Do (Move_Up 1), Do (Move_Down 1)*. The *num* is present in the type definition because *Do (Move_Right n)* is needed by the editor to perform larger moves. For instance, move to right word boundary is achieved by calculating the number of *Move_Right*s required and then performing the *edit_action: Do (Move_Right n*).

The editor can be driven by pressing a function key (which produces a menu on the status line) and then selecting the option wanted by typing the appropriate number. The actual pressing of the function key, e.g. F1, produces *Do (F1 0)* and changes the mode to indicate that a function key has been pressed (*Do (F1 0*)) is the edit_action to place the menu on the status line, similarly for F2-F8). When the user then enters their choice, *get_keyboard* returns *Normal char*. The *edit_action Do (F1 0)* then returns the new *edit_action Do (F1 n)*. (At this point *char* must be a character in the range 0-9. The editor enforces this, *get_keyboard* does not know what mode the editor is in.)

5.3. The *editor* function

The main function *editor* has type:

editor :: edit_action -> state -> (state, output)

where *edit_action* has definition:

```
edit_action ::= Again search_dir |
        Ask text |
        Cerror text |
        Change char |
        Confirm text text reply |
        Copy |
        Ctrl num |
        Cut |
        Noconf text text |
        Delete_word text |
        Delete_Left_Word | Delete_Left_Line | Delete_Left_Doc |
        Delete_Right_Word | Delete_Right_Line | Delete_Right_Doc |
        Do key_action |
```

```
Dump[5] |
Fetch file_name |
Finish reply |
Go direction num |
Goto num |
Insert_word text |
Keep |
Look text search_dir |
Mark |
Message text |
Move_Left_Word | Move_Left_Line | Move_Left_Doc |
Move_Right_Word | Move_Right_Line | Move_Right_Doc |
Normal char |
Paste |
Pop | Push edit_mode |
Redraw |
Find_rep text text | Find_rep_q text text |
Replace text reply |
Search text search_dir |
Store file_name reply |
Unmark
```

these are the editor commands. Note that *editor* does not have an *interact* type. This is because it does not interact with the input stream but rather with *get_keyboard*. *editor* and *get_keyboard* are joined together in the *edit* function:

```
edit :: interact state state
edit (in, st) =  (rem_keyb, current_state, out)
                 where  (rem_keyb, from_keyb, for_screen) = get_keyboard (in, [])
                        (current_state, out) = editor from_keyb st
```

(Note that *from_keyb* is an *edit_action*, which *editor* then applies to it). *edit* has the correct type and can be used with the Thompson combinators.

[5] *Dump* is a special *edit_action* (accessed via typing Ctrl-A) that writes out the current state of the editor to a specified (in the program code) window. It is useful for debugging purposes.

The *edit* function needs to be placed in a "loop" and in the Thompson combinators there is a *while* function:

```
while ::  condition * -> interact * * -> interact * *
while cond inter
      =  whi
         where
         whi = alt cond (inter $sq whi) null
```

and our condition predicate is:

```
continue_editing (in, st) =  if get_mode st ~= End
```

(*get_mode* returns the first item of *edit_mode*) giving:

```
while continue_editing edit
```

as the function that forms the heart of the editor. Before this function is evaluated some screen management must take place and the buffering must be modified (by default all input is buffered until a return is pressed). This leads to the function:

```
edit_prog :: interact state state
edit_prog =
        alt       start_off
                  (seq3    (write initialise_screen)
                           (while continue_editing edit)
                           (alt      stop_on
                                     (write shutdown)
                                     (write "Failed to switch terminal to normal i/o")))
                  (write "Failed to initialise terminal")
```

where *seq3* runs a sequence of three interactions (ensuring that any output appears in the correct order). *start_off*[6] returns *True* if normal keyboard

---

6

```
start_off = begin off
begin :: (text, text, num) -> ([[char]], *) -> bool
begin (s, t, 0) (in, st) = True
```

buffering can be switched off – similarly *stop_on*[1] if it can be returned to normal. *initialise_screen* and *shutdown* send control sequences to the screen to perform various tasks (clear screen, position cursor, etc).

Finally, we use the combinator *run* (from Thompson[2], as modified in section 4) to provide the complete editor function (*ns*):

ns = run edit_prog start_state do_nothing

(*run* requires a function to be applied to the final state and *do_nothing* does just that!). *run* also provides for the input stream to the interactive function *edit_prog* to be standard input.

---

begin (s, t, n) (in, st) = False

stop_on = stop on
*stop :: (text, text, num) -> ([[char]], *) -> bool*
stop (s, t, 0) (in, st) = True
stop (s, t, n) (in, st) = False

where

off = system ttydisable
on = system ttyenable
ttydisable = "stty -echo -icanon min '^A' -isig -ixon <"++tty
ttyenable = "stty echo icanon min '^D' isig ixon < "++tty
tty = getenv "TTY"

ttydisable is the UNIX command string to switch buffering off, *getenv* returns the value of the specified environment variable and *system* returns a 3-tuple comprising of standard output, error output and exit status.

6. The Editor

We set out to build an editor with a functionality and interface that could be used by a "naive" computer user, with one caveat: response times – Miranda is an interpretive system and was never likely to provide response times quick enough to satisfy a user actually using the system. Faster functional languages are available. Miranda was chosen because it was both available and familiar. Also, whereever possible in the programming of the editor, existing commands were re-cycled to avoid writing specific code to perform specific tasks. For instance, the delete left line command is programmed as a mark, move to beginning of line, cut. This approach although defendable in software engineering terms is unlikely to help produce software that has acceptable response times (especially with an interpretive functional language!)

The desire to build an editor that was useful to the naive user was fundamental in the choice to drive the editor through the functions keys (F1-F8) and to implement all the features "advertised" on other keys[7] – delete char, prev (page down), etc. (The keyboard in question being an HP workstation keyboard although any other keyboard could easily be implemented.) When the function keys are used all additional information (filenames, search strings) is prompted for. There is also a mechanism to type in the commands and parameters: this mode of operation is switched to by pressing the Escape key. After this, all text typed up to the next return is dealt with as a command and any missing information is prompted for. Switching to "command" mode also moves the cursor to the bottom line of the screen out of the actual text being edited. The bottom line is also used as a status line (this is where all prompts appear): the status being Insert or Over with the usual meanings. All commands can be abandoned by typing Ctrl-C.

[7] These functions are available via control key combinations for the non-HP keyboard user. See section 6.3.

6.1. Functionality

The actual editing functions that the editor supports are drawn from Sufrin's specification. The following is full description (divided into various categories). These commands can only be typed after typing Esc ([] means optional):

6.1.1. File Management

- **get [<filename>]** – Read file. If filename not supplied then prompt. If file does not exist then issue error message.
- **sa [<filename>] [y/ n/ a]** – Save file. If no filename given get from state. If none in state prompt. If file exists prompt for overwrite permission ask for new name. If "a" abandon command. User can type `sa file.dat y` to automatically overwrite.

6.1.2. Searching & Replacing

All the search and replacing commands can recall their previous arguments.

- **su [<text>]** – Search up
- **sd [<text>]** – Search down
- **au** – Repeat last search up (prompt if search string missing)
- **ad** – Repeat last search down (prompt if search string missing)
- **rp [<text>]** – Replace last found text with <text>
- **fp [<text>] [<text'>]** – Replace text with text' throughout remainder of file. Do not ask. Prompt for any missing information
- **fq [<text>] [<text'>]** – Replace text with text' throughout remainder of file. Ask. Prompt for any missing information

6.1.3. Cut & Paste

- **mk** – Mark text. If used n times marks nth position only
- **ct** – Cut text into buffer (checks for mark)
- **cp** – Copy text into buffer (checks for mark)
- **pt** – Paste from buffer
- **um** – Unmark text

6.1.4. Left Movement & Deletion

- **mlw** – Move left to word boundary (**m**ove **l**eft **w**ord)
- **mll** – Move to beginning of line (also ^b)
- **mld** – Move to beginning of document (also Home)
- **dlw** – Delete to beginning of word
- **dll** – Delete to beginning of line
- **dld** – Delete to beginning of document

Right movement and deletion – As above with r (mrl) with ^e for move to end of line and shift+Home for end of document

6.1.5. Exiting

- **ex [y/ n]** – Exit editor. If there are unsaved changes to the file the user is asked whether to save the file or not (using store command) and then editor exits. User can override with `ex y`.

6.1.6. Miscellaneous

- **re** – Redraw screen.
- **go <n>** – Goto line n.
- **ch [<c>]** – Change delimiter to c. Default delimiters are space or \.

6.2. Function Keys

The function keys give the following menu choices:

F1 1) Get File

2) Save – save file with automatic overwrite (if exists)

3) Save As – save file with new name (or name file)

4) Save & Exit – save file and exit (if file exists overwrite permission requested)

5) Quit – Quit. No questions asked.

F2 Search: 1) -> – Search down
2) <- – Search up
3) Replace
4) Find & Replace
5) Query
Repeat: 6) -> – Repeat last search down
7) <- – Repeat last search up

F3 1) Mark
2) Cut
3) Copy
4) Paste
5) Unmark

F4 Left -- Moves: 1) Word
2) Line
3) Doc
Deletes: 4) Word
5) Line
6) Doc

F5 Right version of F4

F8 1) Goto Line
2) Redraw
3) Change

6.3. Control Keys

The control key in combination with certain characters has meaning:

Ctrl-B: Move to beginning of line.

Ctrl-C: Abandon command! (This cannot be used to abandon, say, a global search and replace operation once it has started because the editor only processes the Ctrl-C when it has finished the previous operation. Ctrl-C does not have its "normal" meaning

because normal keyboard buffering has been switched off by the editor.)

Ctrl-D: Delete Character.

Ctrl-E: Move to end of line.

Ctrl-F: Equivalent to FIND key.

Ctrl-H: Backspace.

Ctrl-I: Insert/Overwrite.

Ctrl-J: Delete Line.

Ctrl-L: Insert Line.

Ctrl-N: Page down.

Ctrl-O: Beginning of document.

Ctrl-P: Page up.

Ctrl-R: Equivalent to REPLACE key.

Ctrl-S: Find next space.

Ctrl-T: End of document.

Ctrl used in conjunction with anything else produces the message "Ctrl n not defined" (n = ASCII sequence number.)

6.4. Operation

The editor is simple enough to start, once in the Miranda environment, simply enter *ns* as the expression to be evaluated[8]. The screen will then clear and “Insert” will appear on the right-hand side of the status line. The editor is now ready to be used. If, say, we now pressed F1 we would see:

```
1) Get File. 2) Save. 3) Save As. 4) Save & Exit. 5) Quit
```

(the remainder of the screen would be blank). We simply select the option we require. If we chose 1, we would be prompted for the file name:

[8] In a UNIX environment, the environment variable TTY must be set to the standard output so that normal terminal i/o can be switched off and on via the *system* function use of the *stty* command (see footnote on page 19).

```
Filename:
```

and so on. When a search and replace operation is being performed the recalled string always appears in the prompt. If ".m" was the last search string, then F2,1 would produce:

```
Search text[.m]:
```

(if there was no previous string then square brackets would contain nothing). To change the string, simply type in a new one and the search will act on that.

7. Discussion

The development of the editor was intended to investigate the programming of a Z specification in a functional language. During this exercise we also took advantage of the opportunity to examine the usefulness of the Thompson combinators and what type of development environment Miranda provided.

7.1. Z specification

Sufrin's specification was developed as a exercise in specifications and not with a view to developing an editor (although it was programmed at the time), the main motivation was to specify a screen editor. In our implementation, as already stated in details in section 2.3, we moved away from the specification for purely pragmatic reasons (a future exercise may well involve a direct translation). We also changed the appearance of the user interface in a number of ways and avoided using the inefficient mirror definition.

Undoubtedly, Z has enormous power because of the precision and clarity that mathematics gives to the specification (although mathematics cannot stop errors or others faults in the specification). But assuming that the specification is correct this still leaves the problem of ensuring that the specification is programmed correctly.

Dealing with faults in a specification has been addressed by techniques like the *me-too*[6] method. *me-too* is an executable specification language that allows specifications to be tested (but *me-too* is only intended for prototyping not developing actual working programs). In a commercial environment this allows the customer to use a working model that if acceptable can then be developed into the final program. This still leaves the development of a correctly working program as a problem but it does help ensure that a working version of the specification can be inspected. This method views the program as the specification. D.A. Turner, the designer of Miranda[7], has

supported this approach in the absence of efficient implementations of functional programming languages.

Clearly, some method of ensuring that a specification can be translated into a usable working program is highly desirable. The ability to relate the specification and the program together in the same fashion as a compiler does for, say, Pascal and machine code should bring major benefits in program reliability and ensuring that specifications are actually met; this would really be a self-documenting program style. Some of the issues involved in translating specifications to programs are discussed in C.B. Jones[8].

7.2. Thompson Combinators

These proved a useful set of functions even though they lacked interactions of the form *get_char* (which we added) and were not quite of the right type. The minor changes and additions we have made do add a good deal to the usefulness, as the additions we made were all driven by the need to develop an interactive application that examined and could respond to every key stroke. Less pleasing was the need use operating system specific code to switch off normal keyboard buffering so that the application could receive every character as it is typed.

These functions undoubtedly make the development of an interactive functional program much simpler. The use of the state also makes quite explicit the data structure that the application is using and by using selector[9] functions to access the state, it can quickly and easily have its structure changed. This gives great flexibility when developing a program to incorporate new ideas or changes – it is an excellent paradigm for programming in any language because it packages up all the components of the application into a single data structure. In an imperative program this

[9] Selector functions are used to extract parts of data structures. Examples are *fst* and *snd* which extract the first and second components of a 2-tuple (i.e. *fst (x, y) = x*)

information is potentially scattered about the various procedures whereas here it is quite clear. (Indeed, the state in the specification never appears explicitly.)

7.3. Debugging and other developmental issues

Developing a reasonably large scale application like a screen-editor does allow the appreciation of a language from not only the elegance of design or implementation view but as a program development tool.

The editor developed here consists of approximately 1,000 lines of program code (i.e. blank lines and comments are ignored). These 1,000 lines are divided up into 20 different files. This, by modern standards, is positively tiny (PC based Windows programs often consist of 100,000's of lines or more of code), but we believe it is large enough to pose problems that allow the examination of the language as a large scale program development tool.

Traditionally languages have some of debugging tool(s) available. The debugger provides various mechanisms to get "inside" the program when it is executing. Developmental/experimental languages do not generally have this type of tool available (this is certainly the case with Miranda), usually because the development of such tools would take considerable time and, initially, are not required: the languages are not going to be used to produce large/commercial programs. In the case of lazy functional languages the reasons are deeper than just the inconvenience of production: the nature of lazy evaluation does not lend itself to normal debugger methods. Unlike an imperative language in which the sequence of evaluation is explicit, with a lazy language evaluations are only done when they are needed and a debugger that evaluated a function for whatever reason could, and probably would, change the order of evaluation. This particular aspect of functional languages is active area of research and a number of alternative debugging methods have been proposed: algorithmic and semantic debugging[9], dataflow analysis[10] and "time travelling"[11]. Investigations in this area may well form part of a future research for us. In particular, we may consider the

development of a set of tools (a workbench) to aid the development and debugging of functional programmes.

7.4. Conclusion

We have shown that a real interactive application can be developed using a lazy functional languages. The resulting program is, at 1,000 lines, compact for the functionality provided (this squares well with the often quoted statistic that imperative language programs are five-to-ten times larger). Further operations could easily be added but this would not alleviate the major drawback: the program is simply far too slow to ever be used by anyone wishing to do some "real" editing – this will remain a problem (for Miranda) until a compiler becomes available. As it stands our work could be viewed as a prototype of an editor that we could now implement using a compilable language or, better still, transform into a compilable language.

8. References

1. B. Sufrin, *Formal Specification of a Display Editor*, Programming Research Group Research Report, Oxford University, 1981.

2. S.J. Thompson, *Interactive Functional Programs: a method and a formal semantics*, UKC Computing Laboratory Report No 48, 1987

3. A. Diller, *Z An Introduction to Formal Methods*, Wiley, 1990

4. J.M. Spivey, *The Z Notation. A Reference Manual*, Prentice-Hall, 1989

5. I. Hayes (Ed), *Specification Case Studies*, Prentice-Hall, 1987

6. H. Alexander & V. Jones, *Software Design and Prototyping using me too*, Prentice-Hall, 1990.

7. D.A. Turner, *Functional Programs as Executable Specifications*, Mathematical Logic and Programming Languages (ed. C.A.R Hoare and J.C Shepherdson), Prentice-Hall, 1985, pp29-50

8. C.B.Jones, *Systematic Software Development using VDM*, Prentice-Hall,1990

9. C. Hall, K. Hammond & J. O'Donnell, *An Algorithmic and Semantic Approach to Debugging*, Functional Programming (Glasgow 1990), Workshops in Computing, Springer-Verlag, Aug 1990, pp44-53

10. D. Sinclair, *Debugging by Dataflow*, Functional Programming, Glasgow 1991,Springer-Verlag, 1991, pp347-351

11. A.P Tolmach & A.W.Appel, *Debugging Standard ML without Reverse Engineering*, Proc ACM Conf on Lisp & Functional Programming 90, June 1990, pp1-12

12. R. Bird & P. Wadler, *Introduction to Functional Programming*, Prentice-Hall, 1987.

13. C. Reade, *Elements of Functional Programming*, Addison-Wesley, 1989.

14. *Miranda System Manual*, Research Software Limited, 1989