Chapter 8

Files and Interaction

8.1 Introduction

This chapter introduces Miranda's mechanisms for the input of data into a program and for the output of data from a program. Data may be transferred between the program and one or more files, or may be transferred between the program and the user. For data transfer to and from the user, the program must take the form of a dialogue; the user must be prompted for information and the program must respond to the user's input. This dialogue between the program and the user is termed *interaction*.

Input and output ("i/o") between programs and files is considered first, starting with simple i/o mechanisms and proceeding to formatted i/o. This is followed by a discussion of interaction with the user via the keyboard and the screen, and finally the mechanisms for interaction with the operating system.[1]

8.2 File input

8.2.1 Simple input

In previous chapters, functions have mainly been applied to data entered at the keyboard; however, this is both tedious and prone to error if large data volumes must be processed. An alternative approach is to store the data as part of the program in the script file, but this entails editing the script file every time there is a change in the data. It is often preferable to put the data into separate files and access these files from within the Miranda system; if the data changes, only the data file need be edited, thereby reducing the chance of introducing an error into the program code. Data can be read from a file by using the built-in **read**

[1]The current version of Miranda is only available for the UNIX operating system (or UNIX clones), and so discussion of operating system interaction will be specific to UNIX.

function, which is of type `[char]->[char]`. The parameter to **read** is a string (that is, a list of characters) representing a UNIX file.[2] The return value is a string containing the contents of the file; if the file does not exist or cannot be read, Miranda halts its evaluation and gives an error message.

The example below shows how the *grep* program, developed throughout this book, can be adapted to work for file input. It uses the built-in function **read** together with the predefined function `lines` which splits a flat string into a list of strings using the newline as a delimiter:

```
in_grep ::  [char] -> [char] -> [[char]]
in_grep regexp filename
    = grep regexp ((lines .  read) filename)
```

Another demonstration of the use of **read** is the following *wordcount* program. This is similar to the standard UNIX facility *wc*, which counts the total number of characters, words and lines in a given file.

In this program the expression `read infile` evaluates (lazily) to a string (a list of char) representing the entire content of the specified input file; this list is used as the input to the auxiliary function `xwc`, which is initialized with zero values in the three accumulators representing the number of lines, words and characters in the input file. Notice that `xwc` does not need to check for any special "end-of-file" marker—the end of the data is represented by the empty list.

```
wordcount ::  [char] -> (num, num, num)
wordcount infile
   = xwc (read infile) 0 0 0
     where
     xwc [] nlines nwords nchars
      = (nline, nwords, nchars)
     xwc (first :  second :  rest) nlines nwords nchars
      = xwc rest nlines (nwords + 1) (nchars + 2),
         if member [' ','\t'] second
      = xwc rest (nlines + 1) (nwords + 1) (nchars + 2),
         if second = '\n'
      = xwc (second :  rest) nlines nwords (nchars + 1),
         otherwise
     xwc (first :  rest) nlines nwords nchars
      = xwc rest nlines nwords (nchars + 1),
         if member [' ','\t'] first
      = xwc rest (nlines + 1) nwords (nchars + 1),
         if first = '\n'
      = (nlines, nwords + 1, nchars + 1),
         otherwise
```

[2]Strictly speaking, it represents a UNIX pathname indicating a file or device.

The characters in the file are read lazily (a character at a time) and are discarded as soon as they have been scanned (this is evident from the fact that none of the alternative return values for `xwc` contain the name `first`).[3]

8.2.2 Formatted input—readvals

The built-in function **read** only returns a list of characters, yet programs often require values of other types. To this end, Miranda provides the **readvals** function, which takes a string representing a file and returns a list of values of a specified type (and so the type of **readvals** is `[char] -> [*]`). Each value in the input file must appear on a separate line and will be returned as a separate item in the list; blank lines and Miranda `||` comments may be embedded in the input file and are *not* read into the resultant list.

UNIX represents the user's screen and keyboard as a file,[4] so that it is possible to read from the keyboard in the same manner as reading from a file (similarly, it is possible to write to a file and have the data appear on the screen). In recognition of this fact, Miranda's **readvals** function checks with the operating system to determine whether the input is actually coming from a file or is coming from a keyboard; if the input comes from a keyboard, **readvals** checks for errors and reacts to bad data by prompting the user to repeat the line (the bad values are omitted from the result list). If the input file is not connected to a keyboard, bad data will cause the program to terminate with an error message.

The following example shows how an input file could be treated as a list of numbers:

```
numberlist ::  [num]
numberlist = readvals "data"
```

Note that Miranda must be able to determine the type of input file used by **readvals**. Thus, despite the polymorphic type of **readvals**, the programmer must specify a monomorphic (non-polymorphic) type each time it is used. To omit the type leads to a compile-time error:

```
numberlist = readvals "data"
```

```
compiling script.m
checking types in script.m
type error - readvals or $+ used at polymorphic type :: [*]
(line 1 of "script.m")
```

[3]In principle, the program should be able to deal with any size of file; unfortunately, with the current implementation, Miranda runs out of space for very large input files.

[4]For example, the file */dev/tty* represents the screen and keyboard currently being operated by the user.

Exercise 8.1
Adapt the *wordcount* program so that it will work for more than one input file.
Exercise 8.2
Explain why the following code is incorrect:

```
wrongsplit infile
           = first second
             where first = hd inlist
                   second = tl inlist
                   inlist = readvals infile
```

User-defined formatting

The **readvals** function operates on the whole file at once and expects all items in the file to be the same type. However, the data file might contain items of different type, in which case it is necessary to treat the file as a list of characters and then to define a collection of functions which will translate a part of the file into data of a different type. The following examples illustrate how to translate data from a list of characters to whatever type is required by the rest of the program. Each user-defined formatting function translates a small portion of data from the start of the file. For each function there is also a data-discarding mirror, which is required in order to access the remainder of the file's data. These functions operate in pairs, much like `hd` and `tl` or `takewhile` and `dropwhile`.

```
>|| user-defined formatting (Page 1 of 2)

read first integer from input file

> readint ::  [char] -> num
> readint infile
>   = string_to_int (takewhile digit (read infile))
>      || where
>      || string_to_int was defined in Chapter 2

drop first integer from input file

> dropint ::  [char] -> [char]
> dropint infile
>   = (dropwhile notdigit (dropwhile digit (read infile)))
```

```
>|| user-defined formatting continued (Page 2)

read first word from input file

> readword ::  [char] -> [char]
> readword infile
>   = (takewhile notspace (read infile))

drop first word from input file

> dropword ::  [char] -> [char]
> dropword infile
>   = (dropwhile isspace (dropwhile notspace (read infile)))

general-purpose character handlers:

> isspace x = member [' ', '\t', '\n'] x
> notspace = (~) .  isspace
> notdigit = (~) .  digit

 digit is built-in:  digit x = '0' <= x <= '9'

readN will read up to a specified number of characters from
its given input file, and result in a tuple containing
a count of the number of characters actually read together
with those characters

> readN ::  [char] -> ([char],num)
> readN infile nchars
>      = (instringLength, instring),
>           if instringLength < nchars
>      = (nchars, take nchars instring), otherwise
>        where
>         instringLength = # instring
>         instring = read infile

> dropN ::  [char] -> ([char],num)
> dropN infile nchars
>      = (instringLength, []),
>           if instringLength < nchars
>      = (nchars, drop nchars instring), otherwise
>        where
>         instringLength = # instring
>         instring = read infile
```

Exercise 8.3
Provide the functions `readbool` and `dropbool` which, respectively, will read and discard a Boolean value from the start of an input file.

8.2.3 Safe input—filemode

So far, the discussion of file input has assumed that the desired input file exists and that the user is entitled to access it. This is clearly not a safe assumption in anything other than a development context. It is possible to check the status of a given file using the built-in **filemode** function, which is of type `[char]->[char]`. The parameter represents a file and the return value represents its UNIX user access permissions.[5]

If the result string is empty then the file does not exist. Otherwise it will contain four characters:

1. The first position will be the character `'d'` if the file is a directory or `'-'` if it is not.
2. The second position will be the character `'r'` if the file is readable or `'-'` if it is not readable.
3. The third position will be the character `'w'` if the file is writable or `'-'` if it is not writable.
4. The final position will be the character `'x'` if the file is executable or `'-'` if it is not executable.

Making use of this information, the *wordcount* program can now be made safe by checking the file status before applying `xwc`:

```
wordcount ::  [char] -> (num, num, num)
wordcount infile
     = error "can't open" ++ infile,
         if cantopen (filemode infile)
            where
             cantopen [] = True
             cantopen ('d' :  rest) = True
             cantopen (any :  '-' :  rest) = True
             cantopen anyother = False
     = xwc (read infile) 0 0 0, otherwise
         || etc
```

[5]The UNIX “group” and “other” access permissions are not available.

8.3 Input and referential transparency

In earlier chapters it has been stressed that the functional programming view of computation is that a name is bound to a value which never changes. This is in direct contrast to the imperative view of programming, where a name is bound to a memory location—the memory location never changes, but the value stored in the memory location can change.

A name in a functional program might be bound to an expression to be calculated, and that expression might be different each time the program is run (because it depends on data entered at run-time); however, once a value has been calculated for that name, it cannot change. This is called "referential transparency".

As a result of referential transparency, the following two definitions must always be equivalent:

```
shared_def g x = y ++ y
                 where
                 y = g x

unshared_def g x = (g x) ++ (g x)
```

Unfortunately, both **read** and **filemode** may exhibit a lack of referential transparency, such that if either were substituted for `g` in the above definitions, and if `x` were the name of a file, the above definitions might not give equivalent results. This is because in a multi-tasking system, such as UNIX, the file contents and the permissions on a file might change at the same time as the Miranda program is running.

In the following example, the function `samecopy` takes a copy of the input file and duplicates it, whereas the function `diffcopy` reads the input file twice. It is just feasible that someone may have edited this file between the first and second evaluations of **read infile**:

```
samecopy infile = instring ++ instring
                  where
                  instring = read infile

diffcopy infile = read infile ++ read infile
```

8.4 File output

Just as it is necessary to take data from the files and the keyboard, it is also necessary to output data to files and the user's screen. In Miranda, all program *output* occurs as part of the value returned by the topmost function, that is, the function whose name is invoked at the Miranda prompt in order to run the program The Miranda on-line manual calls this a "command-level expression".

The following section introduces the concept of *system messages*, which provide mechanisms for a program to control where data is saved and in which format.

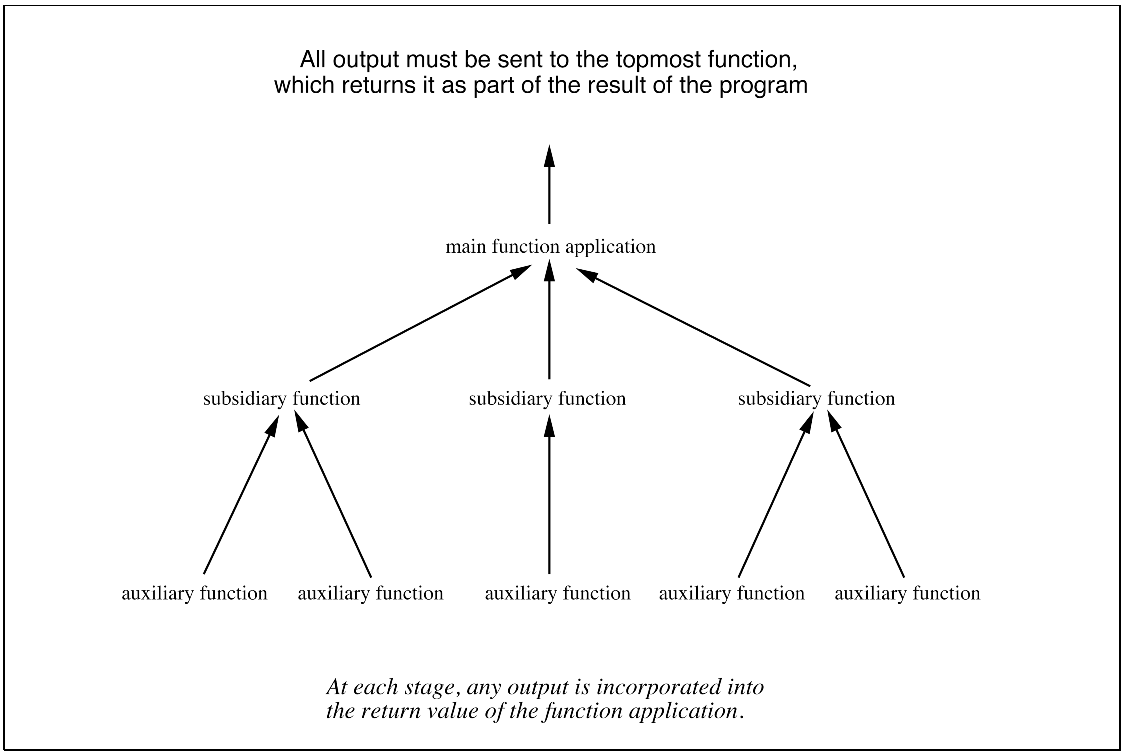
All output must be sent to the topmost function,
which returns it as part of the result of the program
main function application
subsidiary function
subsidiary function
subsidiary function
auxiliary function
auxiliary function
auxiliary function
auxiliary function
auxiliary function
At each stage, any output is incorporated into
the return value of the function application.

Figure 8.1 File output.

8.4.1 System messages

The value of a command-level expression is always a list of system messages, that is, it has result type `[sys_message]`. A system message is defined as an algebraic type, where each constructor corresponds to a different output action:

```
sys_message ::= Stdout [char] | Stderr [char] |
                Tofile [char] [char] | Closefile [char] |
                Appendfile [char] | System [char] | Exit num
```

For example, a program might produce the result:

```
[ Stdout "twinkle", Tofile "myfile" "twinkle little star" ]
```

The above list of system messages is a sequence of commands to the Miranda system; these commands are obeyed in left-to-right order. Thus, in the above example, the Miranda system will first send the string `"twinkle"` to the standard output (usually the user's screen), then send the string `"twinkle little star"` to the file `myfile`. The actions of each of the `sys_message` constructors will be explained in detail in the following subsections.

The default message wrapper

The above discussion may seem somewhat disconcerting, since none of the example programs in the preceding chapters have required the top-level function of a program to produce a value of type `[sys_message]`. This is because they have all operated under a "default wrapper", which converts simple program output into a value of type `[sys_message]`. The default wrapper assumes that the output is meant to be displayed as a string on the screen. If the program produces the value `x` then it will silently be translated into `[Stdout (show x)]`. This works for a value of any type recognized by **show**.

This feature is particularly useful to enable rapid prototyping and separate module testing.

8.4.2 Writing to a file

So far, all the functions in this book have output their results to other functions or to the screen; hence the output data has not survived the Miranda session. It is easy to save this output data in a file using the `sys_message` constructor `Tofile` which takes a string representing the name of the output file (or device, such as a printer) and a string consisting of the output data.

The first time an element of the form *Tofile file data* appears in the `sys_message` list, the *file* is opened for writing and *data* is written to it: subsequent use of the same `Tofile` expression (most likely with new data) will cause the new data to be *appended* to the *file*. Note that, in keeping with the spirit of UNIX, if the *file* already exists then its old contents will be deleted before the new data is first written.

The following example copies the contents of a named input file to a named output file, discarding all "white space" characters. This is achieved by reading the entire input file, then removing all white space characters and then writing the result:

```
compresscopy ::  [char] -> [char] -> [sys_message]

compresscopy infile outfile
    = [Tofile outfile (xcompress (read infile))]
      where
       xcompress = filter (( ) .  isspace)
       isspace x = (x = ' ') \/ (x = '\t') \/ (x = '\n')
```

Appending to files

Sometimes, it is necessary to add to an existing file rather than write a new file. This can be achieved using the constructor `Appendfile`, which takes a string representing the name of a `file` that will have data concatenated to it. This element must appear in the `sys_message` *before* any `Tofile` element that writes to `file`.

```
add_to_diary ::  [char] -> [sys_message]

add_to_diary message
         = [Appendfile "diary", Tofile "diary" message]
```

8.4.3 Formatted output

Data values other than character lists, can be written to output files using **show**. For example:

```
writeint ::  [char] -> num -> [sys_message]

writeint outfile n = [Tofile outfile (show n)]
```

Two restrictions apply:

1. Infinite data values cannot be written to a file.
2. The **show** keyword only works for known types. It will not work for functions, and abstract types require the special treatment described in Chapter 7.

If **show** did not exist then it would be necessary to write separate functions to convert values to character lists; thus the above example could be written as:

```
writeint ::  [char] -> num -> [sys_message]

writeint outfile n
    = [Tofile outfile (int_to_string n)]
      || using int_to_string as shown in Chapter 2
```

Notice that any data written to a file using **show** is stored in a form that is appropriate for subsequent reading by using **readvals**. However, it is generally considered bad style to have the same file open for both reading and writing during a program. This is because of the danger of accidentally overwriting data that has not yet been read.

Closing files

The constructor `Closefile` takes a string giving the name of a file and, assuming the file has previously been opened by a `Tofile` message, it will close that file.

In practice, explicit file closing is normally unnecessary because all files are normally closed on exit from the Miranda prompt. Nonetheless, there are three situations where it is necessary:

1. Where it is required to read from and write to the same file within a single program; it is normally necessary and advisable to close the file after writing and before reading (or vice versa).
2. Where the number of files open exceeds the operating system limits.
3. Where it is required to flush the output to the file, to ensure all data has been written.

8.5 Special streams

Miranda i/o was designed with the UNIX operating system in mind, and follows the UNIX approach of having three "special" files, the standard input (normally representing the user's keyboard), the standard output (normally the user's screen) and the standard error output (also normally the user's screen). These special files are automatically opened at the start of any UNIX process which has been started from the standard UNIX command interpreter (the "shell") and provide convenient communication with the terminal. Thus, a Miranda session will always start with these three files already open: the standard input is open for reading, the standard output and standard error output are both open for writing.

These three special files are sometimes called "streams" and are primarily used for *interactive* programming, as will be discussed later in this chapter. They may however also be redirected (at the UNIX command line), so that they are connected to files instead of being connected to the user's terminal; they may therefore be used to prototype file processing programs (input and output can initially be with the keyboard and screen, and then redirected to files once the program is working correctly).

8.5.1 Standard input

The special form `$-` is a list of characters representing the contents of the standard input. Multiple occurrences of `$-` always represent the same input. Hence `$- ++ $-` reads data from the keyboard and duplicates it. Keyboard data entry is terminated by entering `<control-D>` *at the start of a new line.*

Formatted standard input

The use of `$+` is the equivalent of using **readvals** with the name of a file denoting the standard input. For example:

```
Miranda foldr (+) 0 $+
```

will take a a sequence of numbers from the keyboard (one per line) up to the next `<control-D>` and then return their sum.

8.5.2 Standard output and standard error output

To send data to the standard output stream, the program should include the system message `Stdout x` as part of its result, where `x` is a value of type `[char]`. Similarly, data can be sent to the standard error output, using the system message `Stderr x`. If it is necessary to send a number to the standard output or standard error output streams, it should first be formatted as a string:

```
Miranda [Stdout "hello"]
hello

Miranda [Stderr (show (300 + 45))]
345
```

In each of the above examples the result will appear on the user's screen. Sometimes it is necessary to clear the screen before displaying messages, this will be explained in the next subsection.

The "System" message

It is possible to send commands to the UNIX operating system by using `System` messages as part of the output of the program. This message takes a string argument which is interpreted and evaluated by the UNIX command interpreter. For example, the program can clear the user's screen by using the message `System "clear"` as part of its `sys_message` list. Any valid UNIX command can be given:

```
Miranda [System "date"]
Sat May 14 14:51:07 BST 1994

Miranda [Stdout "The date is: ", System "date", Stdout " A.D."]
The date is: Sat May 14 14:51:07 BST 1994
A.D.
```

Note that in the last example the date printed by the system includes a newline: thus, the characters `" A.D."` appear on the next line.[6] Finally, notice that the command evaluated by the operating system might not produce any visible output:

```
Miranda [Stdout "Start:", System "mv a b", Stdout ":End"]
Start::End
```

The "Exit" message

When a UNIX process terminates it may return an exit status number to the operating system. This can be achieved with the system message `Exit x` where `x` is a number between 0 and 127. As soon as `Exit` is detected in the output of the program, the program is terminated. Thus, any system messages following `Exit` will be ignored. An exit value of zero usually indicates that the program has terminated with no errors.

If a program ends without using the `Exit` system message, Miranda will generate an exit value of `0` for normal exit and `1` for program exit due to an error.

8.6 Interaction

Interactive programs must interact with the user's terminal—to accept input from the keyboard and to print output to the screen. Most terminals provide a character-based interface to the program; that is, the keyboard produces characters and the screen will accept characters. Since there is no way to tell in advance how many characters a user will type at the keyboard, Miranda treats the data coming from the keyboard as a (potentially infinite) list of characters; output is treated similarly. Thus, an interactive Miranda program should have the type `[char] -> [char]`.

The `[char] -> [char]` paradigm is simple and convenient. However, it is necessary to keep in mind the following two points:

1. As explained previously with regard to file output, *all* output must be channelled through the topmost function.
2. It is necessary for the programmer to specify the precise order in which user interaction should take place.

[6] If it is required to format the output from a UNIX command then the program should use the `system` function discussed in Section 7 of this chapter.

8.6.1 Specifying a dialogue sequence

The user dialogue is one of the few places[7] in a Miranda program where the specification of evaluation order is necessary; in the rest of the program it is normally sufficient merely to specify how results will be combined to produce a new value and leave the evaluation sequence to Miranda.

Miranda determines the evaluation order by inspecting the data dependencies implied by the program. For example, in the expression `((3 * 4) + 5)` the addition operator cannot proceed until the values of both of its operands are known and so the multiplication must take place before the addition. In the example `(2 + (fst ((3 * 4), (5 / 6))))`, the function `fst` does not need to know the result of the division and so the division is never done (this is lazy evaluation); however, the addition needs to know the result of the multiplication and so the multiplication is done before the addition. Finally, note that the Miranda system is free to choose an evaluation order at random where there are alternatives. For example, in the expression `((4 * 5) + (6 * 7))` both multiplications must be done before the addition, but which multiplication happens first is not defined. Normally, the Miranda programmer need never be concerned about such operational issues, but it becomes important when dealing with the interactive part of a program, as shown in the rest of this section.

There are two simple ways in which a programmer can force Miranda to evaluate one expression before another:

1. Use function composition. For example, in the expression `((f . g . h) 45)` the application of the function `h` will be evaluated first, followed by the application of `g` to the intermediate result, followed by `f`. Lazy evaluation will still mean that only the necessary parts of the data will be evaluated, but now the programmer has some control over the order in which function applications are considered.
2. Use pattern matching, since if an argument must be checked against any pattern other than a formal parameter name then it must be evaluated at least enough to determine whether it matches, and furthermore this evaluation must happen *before* the function body is evaluated.

Simple interaction example

The following simple example demonstrates a dialogue with the user where the program waits for the user to type a line and then prints the date; it does this repeatedly until the user types `<control-D>`. The correct sequence of question and response is guaranteed by the use of pattern matching:

[7]Control of evaluation order may be necessary for advanced program optimisation (which is beyond the scope of this book) and perhaps where the program interacts with the outside world using `read` or `system`—see Section 8.7.

```
prompt = "Please enter something:  "
msg = "Here is the date:  "

loop [] = [Stdout "\nGoodbye"]
loop ('\n':rest)
     = Stdout ("Please press just one character, "
       ++ "followed by the Return key")
       :  (loop rest)
loop (any:'\n':rest
     = Stdout msg :  System "date"
       :  Stdout prompt :  (loop rest)
loop (any:rest)
     = Stdout ("Please press just one character, "
       ++ "followed by the Return key")
       :  (loop rest)

main = Stdout prompt :  (loop $-)
```

By contrast, the following example gets the sequencing wrong and prints the first date before it is requested:

```
prompt = "Please enter something:  "
msg = "Here is the date:  "

wrongloop ip
     = Stdout msg :  System "date"
       :  Stdout prompt :  (rest ip)
       where
       rest [] = [Stdout "\nGoodbye\n"]
       rest (x:xs) = wrongloop xs

main = Stdout prompt :  (wrongloop $-)
```

The above error is quite understandable, since the output of the program does not depend on the value of the data entered by the user. This highlights the difference between programming the interactive part of a program (where the correct sequence of operations is of primary importance) and the body of the program (where values are of primary importance).

8.6.2 Using a menu

A common user interface is to present the user with a menu of items and ask the user to choose one item (perhaps an action) by entering a number. A simple example of this behaviour is illustrated in the following program which mimics a drink-vending machine. Of course, there are no real drinks on offer; it is a “virtual” vending machine. This program will be presented as four different versions: the first two will illustrate two equivalent ways to implement a single menu and the second two will illustrate two equivalent ways to implement a more complex program with three menus.

The user interface for the program with just one menu is simple:

1. First, the user is presented with a choice of either Tea, Coffee, Soup or Quit. The user makes a choice by entering a number (1, 2, 3 or 4) and the computer prints an acknowledgement on the screen.
2. If the choice was 4 (for Quit) the program stops. Otherwise, it loops and prints the menu again.

The above interaction between human and computer will be the same in both of the following two examples. The common code shared between the two examples is now presented:

```
Screen messages for first two vending program examples:

> welcome :: [char]
> welcome
>   = "Welcome to the Virtual Vending Machine\n\n"
>     ++ "You may choose from the following menu:\n\n"
>     ++ "1.  Tea\n2.  Coffee\n3.  Soup\n"
>     ++ "4.  Quit\n" ++ request

> request :: [char]
> request
>   = "Please enter the number of your choice:  "

Message to confirm the user's choice:

> confirm :: [char] -> [sys_message]
> confirm choice
>   = [ Stdout ("\nThank you for choosing "
>               ++ choice ++ ".\n")]

Leave the program, with ok exit status:

> quit :: [sys_message]
> quit = [Stdout "End of program\n", Exit 0]
```

Preformatted menu input

If it is clear that the user will always enter values of the same type (for example, if the user can only enter numbers in response to menus) then it is appropriate to use the $+ special form to represent the input. Miranda will automatically interpret the characters typed at the keyboard and translate them into values of the correct type. Remember that the user must always press the Return key to send the input to the program and if the user enters a value that cannot be interpreted then Miranda will issue a warning and wait for input of the correct type.

```
>|| version 1 of vending program

> vend ::  [sys_message]
> vend = (System "clear" ) :  dialogue $+

> dialogue ::  [num] -> [sys_message]
> dialogue ip
>    = [Stdout welcome] ++ (next sel)
>      where
>      (sel :  rest) = ip
>       next 1 = confirm "tea" ++ dialogue rest
>       next 2 = confirm "coffee" ++ dialogue rest
>       next 3 = confirm "soup" ++ dialogue rest
>       next 4 = quit
```

In the above example, pattern matching on the input is done inside the **where** block rather than in the formal parameter list for the function; this ensures that the welcome message is printed *before* the program interrogates the keyboard. It is instructive to run the following (wrong) version which will wait for user input before printing the menu:

```
> vend = (System "clear") :  wrongdialogue $+

> wrongdialogue (sel :  rest)
>    = [Stdout welcome] ++ (next sel)
>      where
>       next 1 = confirm "tea" ++ wrongdialogue rest
>       next 2 = confirm "coffee" ++ wrongdialogue rest
>       next 3 = confirm "soup" ++ wrongdialogue rest
>       next 4 = quit
```

Exercise 8.4
Explain why the following attempt at `vend` is incorrect:

```
vend = (System "clear") : wrongdialogue $+
wrongdialogue ip = [Stdout welcome] ++ quit, if sel = 4
                 = [Stdout welcome] ++ (next sel), otherwise
                   where
                   (sel : rest) = ip
                   next 1 = confirm "tea" ++ wrongdialogue rest
                   next 2 = confirm "coffee" ++ wrongdialogue rest
                   next 3 = confirm "soup" ++ wrongdialogue rest
                   next 4 = quit
```

Program-formatted menu input

If the program expects the user to enter values of differing type in any order, then it is necessary to use the `$-` special form to represent the input. The user input will be available to the program as a list of characters which can then be interpreted according to the types of values expected. Remember that the user must always press the Return key to send the input to the program. When using `$-`, the end-of-line character representing the Return key will also be part of the input list of characters, and so the program must interpret this Return character.

The following version of `vend` employs `$-` to allow the user to enter words representing the desired drinks or to enter numbers representing the menu option. Notice that the `split` function discards the end of line character.

```
>|| version 2 of vending program

> vend :: [sys_message]
> vend = (System "clear") : dialogue $-

> dialogue :: [char] -> [sys_message]
> dialogue ip
>   = [Stdout welcome] ++ (next sel)
>     where
>     (sel,rest) = split ip
>      next "1" = confirm "tea" ++ dialogue rest
>      next "tea" = confirm "tea" ++ dialogue rest
>      next "2" = confirm "coffee" ++ dialogue rest
>      next "coffee" = confirm "coffee" ++ dialogue rest
>      next "3" = confirm "soup" ++ dialogue rest
>      next "soup" = confirm "soup" ++ dialogue rest
>      next anyother = quit
>
>      split [] = ([],[])
>      split ('\n' : rest) = ([],rest)
>      split (x : rest) = (x : a, b)
>      where
>        (a,b) = split rest
```

Plumbing multiple menus together

This third version of the vending machine program and the final version both have more than one menu. It will be seen that the correct sequencing of the dialogue between multiple menus requires the careful “plumbing” of both the user’s input and the program’s output.

These programs share an interface consisting of three menus:

1. Initially, the user is presented with a choice of either Tea, Coffee or Soup. The user makes a choice by entering a number (1, 2 or 3) and Miranda prints an acknowledgement on the screen.
2. If the user has chosen Tea or Coffee (but not if the user has chosen Soup), a second menu allows the user to choose Milk, Sugar, Milk and Sugar, or None. The user makes this choice by entering a number (1, 2, 3 or 4) and Miranda prints an acknowledgement on the screen.
3. Finally, the computer displays a message on the screen, followed by a menu which allows the user to choose either to exit the program or to choose another drink.

```
Screen messages for third & fourth vending programs:

> welcome :: [char]
> welcome = "Welcome to the Virtual Vending Machine\n\n"
>           ++ "You may choose from the following menu:\n\n"
>           ++ "1.  Tea\n2.  Coffee\n3.  Soup\n"
>           ++ "4.  Quit\n" ++ request

> request :: [char]
> request = "Please enter the number of your choice:  "

> menu2 :: [char]
> menu2 = "You may also choose from:\n" ++
>         "1.  Milk and Sugar\n" ++
>         "2.  Milk only\n" ++
>         "3.  Sugar only\n" ++
>         "4.  None of the above\n" ++ request

> menu3 :: [char]
> menu3 = "Enjoy your virtual drink!\n\n" ++
>         "Please enter 1 to exit or 2 for another drink\n"

Message to confirm the user's choice:

> confirm :: [char] -> [sys_message]
> confirm choice = [Stdout ("\nThank you for choosing "
>                  ++ choice ++ ".\n")]

Leave program with ok exit status, discarding surplus input

> quit :: [num] -> [sys_message]
> quit x = Stdout "End of program.\n" : [Exit 0]
```

The rest of the program for version three is now presented. The `dialogue` function prints the first menu to the screen as before, and inspects the user's response by pattern matching on the input; this is done inside the **where** block in order to achieve the correct sequencing. However, if the user has chosen Tea or Coffee then a second menu must be displayed and the input must be further interrogated. Once this has been achieved, the `dialogue` function must present the user with a third menu and, according to the user's response, either terminate or recurse.

The third menu interaction is achieved in a straightforward manner by the function `check` inside the **where** block. The second menu interaction is, however, more complex because it involves the action of functions which are *not* part of the `dialogue` function. In general, it is not possible to know how much of the user input will be consumed by these separate functions and so the entire user input data must be transferred explicitly *into* the subsidiary function (this is sometimes called “downward plumbing”). Subsequently, the input data which has not been consumed must be transferred *out of* the subsidiary function as part of its result for further inspection by the `dialogue` function (this is sometimes called “upward plumbing”).

In this example, the functions for tea, coffee and soup return the remainder of the input as part of their result tuple and the other part is the output to be sent to the user. This is another example of “upward plumbing”, and it is important that the `dialogue` function should correctly sequence this data as part of the overall output of the program.

```
>|| version 3 of vending program (Page 1 of 2)

> vend :: [sys_message]
> vend = (System "clear") : dialogue $+

> dialogue :: [num] -> [sys_message]
> dialogue ip
>   = [Stdout welcome] ++ next ++ (check rest2)
>     where
>      (sel : rest) = ip
>      (next, rest2)
>          = ([tea, coffee, soup] ! (sel - 1)) rest
>      check xip = [Stdout menu3] ++ xcheck xip
>                  where
>                  xcheck (1:rest3) = quit rest3
>                  xcheck (2:rest3) = dialogue rest3
>                  xcheck any       = quit any
```

```
>|| version 3 of vending program continued (Page 2)

> tea :: [num] -> ([sys_message], [num])
> tea ip
>  = (confirm "Tea" ++ [Stdout menu2] ++ next, rest)
>    where
>     (sel : rest) = ip
>     next = acknowledge "Tea" sel

> coffee :: [num] -> ([sys_message], [num])
> coffee ip
>  = (confirm "Coffee" ++ [Stdout menu2] ++ next, rest)
>    where
>     (sel : rest) = ip
>     next = acknowledge "Coffee" sel

> soup :: [num] -> ([sys_message], [num])
> soup ip = (confirm "Soup", ip)

> acknowledge :: [char] -> num -> [sys_message]
> acknowledge d 1 = confirm (d ++ " with Milk & Sugar")
> acknowledge d 2 = confirm (d ++ " with Milk")
> acknowledge d 3 = confirm (d ++ " with Sugar")
> acknowledge d 4 = confirm d
```

Exercise 8.5
Why is it necessary for `check` to have a **where** block, and why is `confirm` repeatedly applied within `acknowledge`?

Menus using continuation functions

The final version of the vending machine program uses a general-purpose function which takes the input stream, a message and a list of functions; it prints the message and reads the user's input (which must be a number), and then applies one of the functions to the remainder of the input stream. The function which is applied (often called a "continuation function", because it determines how the program will continue) is chosen according to the number returned by the user. Note that the whole user interface is mutually recursive; it is not easy to reason about and is difficult to test, since individual functions cannot be tested in isolation from the other functions.[8] However, this style of programming user interfaces is enthusiastically promoted and supported by some other functional programming languages; it can also be generalized to encompass interaction with the operating system, with different continuation functions provided for successful and unsuccessful operations. This style of programming is sometimes known as “continuation-passing style”, or just “CPS”.

```
>|| version 4 of vending program

>vend :: [sys_message]
>vend = (System "clear") : dialogue $+

>dialogue :: [num] -> [sys_message]
>dialogue ip
>  = gendialogue ip welcome [tea, coffee, soup]
>    where
>     tea newip = xdial "Tea" newip
>     coffee newip = xdial "Coffee" newip
>     xdial d newip
>       = confirm d ++ gendialogue newip menu2 (extras d)
>     extras d = map option [(d," with Milk & Sugar"),
>                            (d," with Milk"),
>                            (d," with Sugar"), (d,"")]
>     soup newip = confirm "Soup" ++ continue newip

>option :: ([char],[char]) -> [num] -> [sys_message]
>option (drink, extra) ip
>   = confirm (drink ++ extra) ++ continue ip

>continue :: [num] -> [sys_message]
>continue ip = gendialogue ip menu3 [quit, dialogue]

>gendialogue :: [num] -> [char] -> [[num]->[sys_message]]
>               -> [sys_message]
>gendialogue ip msg fns
>   = Stdout msg : xdial ip fns
>     where
>      xdial [] fns = quit []
>      xdial (x : rest) fns = (fns ! (x - 1)) rest
```

[8]The reader might notice that it implements a finite state machine (Minsky, 1967).

8.7 Advanced features

This section brings together various advanced Miranda features which either facilitate interaction with the operating system or provide greater control over the evaluation mechanism. Because all current implementations of Miranda are designed to run on the UNIX operating system (or a UNIX equivalent such as LINUX), the operating-system features discussed in this section are specific to UNIX.

8.7.1 Interaction with the Miranda evaluation mechanism

The Miranda lazy-evaluation mechanism provides a powerful computational model. However, it is sometimes useful to be able to encourage Miranda either to evaluate two expressions in a certain order, or to evaluate an expression more fully than it would otherwise. This manipulation of the evaluation mechanism is necessary either in the user-interface part of a program, in order to achieve a desired sequencing effect, or in the main body of a program, in order to optimize the efficiency of the code.[9]

In addition to function composition and pattern matching, Miranda offers two built-in functions:

1. The built-in function `seq`. This function has type:

   ```
   seq:: * -> ** -> **
   ```

   The `seq` function takes two arguments. It checks that the first argument is not completely undefined, which requires some evaluation of the first argument, but not full evaluation. It then returns the second argument as its result, so that the extent to which the second argument is evaluated depends on the context in which `seq` has been applied.
   The phrase “not completely undefined” means that if the first argument is a list then it will be evaluated to the extent that its length is known but its elements may still be undefined.
2. If the `seq` function does not provide sufficient evaluation of the first argument, it can be combined with the built-in function `force`. This function has type:

   ```
   force:: * -> *
   ```

   The `force` function forcibly evaluates all of its argument and then returns that argument’s value as its result. Thus, `force` cannot be used on its own to enforce order of evaluation of one thing before another, but it can be used in conjunction with `seq`:

   ```
   fullseq x y = seq (force x) y
   ```

[9]However, issues of efficiency are beyond the scope of this book.

The functions `seq` and `force` evaluate their arguments to differing extents: `force` will return an error (or an undefined result) if evaluation any part of its argument returns an error (or is undefined), whereas `seq` can sometimes return a result which contains undefined parts. For example:

```
Miranda seq [(3 div 0)] 45
45

Miranda force [(3 div 0)]
[
program error: attempt to divide by zero
```

It is possible to get correct dialogue sequencing using `seq` instead of pattern matching. In the following example, the use of `seq` ensures that the first item of the standard input is evaluated before the program produces the next date:

```
prompt = "Please enter something:  "
msg = "Here is the date:  "

loop [] = [Stdout "\nGoodbye"]
loop ip
   = seq (hd ip) (Stdout msg :  System "date"
                  :  Stdout prompt :  (loop (tl ip)))

main = Stdout prompt :  (loop $-)
```

8.7.2 Interaction with UNIX

Subsection 8.5.2 showed how the *System* message could be utilized to ask the operating system to evaluate a command; any output from that command is printed to the standard output. This subsection shows how the output from an operating system command can be manipulated *inside* a Miranda program.

The built-in function `system` takes a string argument which is a command to be evaluated by the UNIX command interpreter. A new UNIX process is created[10] in order to run this command and the result of the `system` function is a three-tuple containing:

1. A string containing whatever data the program wrote to its standard output. This list of characters is created lazily—each character is available as soon as it is output by the program.
2. A string containing whatever data the program wrote to its standard error output. Each character in the list is available as soon as it is output by the program.

[10]The new process has its standard input closed, so that it cannot interfere with input to the Miranda program.

3. A number containing the exit status of the program (0 means that the program terminated correctly; any other value indicates that an error occurred). The number will always be an integer between 0 and 127. The number is only available after the program has finished.

The type of this function is therefore:

```
system :: [char]->([char],[char],num)
```

Note that Miranda's lazy evaluation of the first two elements of the above tuple means that it is possible for the Miranda program and the called program to run concurrently, with synchronizing pauses only necessary if the Miranda program tries to read data faster than the called program can generate that data.

If UNIX cannot evaluate the given command, the result returned by `system` will be `([], error_message, -1)`, where `error_message` is some error message, indicating why the command failed.

Referential transparency and system

The function `system` provides a general mechanism for interfacing with the operating system. It is perhaps rather too powerful than is appropriate for a functional language. Recall that in a functional language one attempts to "program by value" rather than "program by effect", and it is precisely this value-oriented discipline that gives functional programs enormous advantages over imperative programs; functional programs tend to be shorter, more modular and easier to understand.

With the `system` function it is possible to introduce the "program by effect" style into a Miranda program; *this should be avoided*! The `system` function should only be used to gain information from the operating system, and never as a mechanism to cause some effect on the system outside of the Miranda program. It should be remembered that a functional program should only effect the rest of the system by means of system messages in the result of the program. Thus, to have an effect on the operating system, one should use the *System* message. By contrast, to gain information from the operating system one should use the `system` built-in function.

Despite the above discussion, some UNIX commands will by their very nature introduce a degree of referential opacity (the opposite of referential transparency) to a Miranda program. For example:

```
shared_def g x = y ++ y
                 where
                 y = g x

unshared_def g x = (g x) ++ (g x)

main = shared_def system "date"
```

The program above will read the date (which includes the current time) once and print out that value twice. However, if the program were changed so that the `main` function called the function `unshared_def` instead of `shared_def` then the date would be read twice and it is quite likely that the second reading would be different from the first. This demonstrates that `system` is not referentially transparent and should be used with great care!

8.7.3 Modifying UNIX interactive behaviour

In all the previous examples, the user has been required to press the Return key in order to send input to the computer program. Furthermore, every time the user presses a key on the keyboard this is displayed on the screen; this is done by UNIX, not by the computer program.

UNIX allows these two behavourial features (and many others) to be controlled by the program. This permits the programmer to develop a more direct interaction between the program and the user, as illustrated by the following simple program which provides a square board on the screen within which the user can manoeuvre. This simple form of interaction is the basis of many computer games, though issues of optimized screen control, the use of bit-mapped graphics and windowing systems are beyond the scope of this book.

In the following example, the system message `System "stty cbreak -echo"` contains the two instructions to UNIX:[11]

1. Allow each keyboard character to be input to the program when it is pressed (that is, do *not* wait for the Return key to be pressed before sending characters to the program).
2. Do not echo the keyboard character to the screen. In this example, this is just an aesthetic decision concerning screen display; however this feature is required for many applications, such as full-screen editors and to conceal password entry.

Note that at the end of the program, UNIX is instructed to return to "normal" behaviour by use of the system message `System "stty -cbreak echo"`. Also note that the modification of UNIX behaviour in this way is not suitable for the user once the program has finished and so the programmer must be sure that the program resets this behaviour *whenever the program terminates, for whatever reason.* The programmer should be particularly careful of the following causes of program termination:

1. Normal termination due to the end of input.
2. Normal termination due to some other reason (for example, the end of the game, if the program implements a game).

[11]This `stty` command will not necessarily work correctly for all versions of UNIX; readers are recommended to consult their local system documentation.

3. Termination due to a program-detected error (thus, use of the `error` function becomes more complex, as demonstrated in the example below).
4. Termination due to an error in the program being detected by Miranda (thus, the program should be thoroughly tested, especially for missing cases and the possibility of list indexes exceeding the bounds of the list, before being released to the user).

```
>|| Boardgame program:  places 'X' onto board (Page 1 of 2)

> main = [System "stty cbreak -echo"]
>        ++ (display board) ++ (dialogue board startpos $-)

> board = Board (rep 10 " ")
> startpos = (5,5)

> dialogue bd (x,y) ip
>  = xdial bd ip
>    where
>     xdial Error any = dialogue_over "Game error\n" 1
>     xdial b ('n' :  rest) = newpos (0,-1) rest
>     xdial b ('e' :  rest) = newpos (1,0) rest
>     xdial b ('w' :  rest) = newpos (-1,0) rest
>     xdial b ('s' :  rest) = newpos (0,1) rest
>     xdial b ('q' :  rest) = dialogue_over "Game over\n" 0
>     xdial b any = dialogue_over "Game error\n" 1
>
>     newpos (p,q) ip
>       = (display (setboard bd (x+p,y+q)))
>          ++ dialogue (setboard bd (x+p,y+q)) (x+p,y+q) ip

> dialogue_over m s
>     = [Stdout m, System "stty -cbreak echo", Exit s]

> abstype game_board
> with
>  board    ::  game_board
>  setboard ::  game_board -> (num,num) -> game_board
>  display  ::  game_board -> [sys_message]
>  dialogue ::  game_board -> (num,num) -> [char]
>                 -> [sys_message]

> board_type ::= Error | Board [[char]]
> game_board == board_type
```

```
>|| Boardgame program continued (Page 2)

> setboard Error (x,y) = Error
> setboard (Board bd) (x,y)
>   = Error, if (x<0) \/ (x>9) \/ (y<0) \/ (y>9)
>   = Board (take y bd ++ [setrow (bd!y) x]
>            ++ drop (y+1) bd), otherwise
>     where
>      setrow row x
          = (take x row) ++ "X" ++ (drop (x+1) row)

> display Error = [System "clear",Stdout "ERROR\n"]
> display (Board bd)
>    = [System "clear",Stdout (lay newbd)]
>      where
>       newbd = [rep 12 '-'] ++
>               (map (++ "|") (map ('|' :)  bd))
>               ++ [rep 12 '-']
```

Exercise 8.6
Use the above board as the basis for a simple game of noughts and crosses (tic-tac-toe).

8.8 Summary

This chapter introduced tools to facilitate communication between the programmer and the world outside of the Miranda system. File input makes it possible to have programs that work on different data values, without artificially amending the script file; whilst file output allows the results of Miranda programs to be saved outside of a Miranda session. The chapter continued by introducing special files or streams for communicating with the keyboard and the screen, hence allowing for interaction with another user. The process of interaction requires careful consideration of the sequencing of function application via pattern matching or the use of continuation functions. Finally, some mechanisms for interacting with the operating system were discussed.