Part I
Chapter 5
Input and Output

5.1 Changing the world
5.2 Combination of input/output functions
5.3 Some simple dialogs
5.4 A simple window
5.5 Timers
5.6 A line drawing program
5.7 Exercises

In this chapter it is described how interactive functional programs can be written. This chapter uses the (to Clean 1.1 converted) 0.8 version of the I/O library.

In the previous Chapter we introduced the two key techniques for interactive functional programming:

- environment passing which is used to control the order in which I/O is performed;
- the environment can be passed in such a way that the uniqueness of references is guaranteed, such that destructive updates can be used (in that case the technique is also called *state-transition*).

5.1 Changing the world

Suppose we want to write a program that copies a complete file. It is easy to define an extension of the examples in the previous section such that not just one or two characters are written but a complete list of characters is copied to a file. Combining this with a (as yet unspecified) function to read characters from a file gives a function to copy a file.

```
CharListWrite :: [Char] *File -> *File
CharListWrite []     f = f
CharListWrite [c:cs] f = CharListWrite cs (fwritec c f)

CharFileCopy :: File *File -> *File
CharFileCopy infile outfile = CharListWrite (CharListRead infile) outfile
```

The function `CharListWrite` is left-recursive and strict such that its machine code will be very close to the code of a classical loop.

Reading characters from a file requires a few more lines than writing since for reading not only an environment (the file, i.e. the pointer in the file to the next character to read) has to be passed but also a result has to be yielded. The file from which characters are read is not required to be unique since no destructive update is involved in reading.

The read function is lazy. So, character by character the file will be read when the characters are needed for the evaluation (the actual library implementation of `sfreadc` will probably use some kind of buffering scheme).

```
CharListRead :: File -> [Char]
CharListRead f
    | not readok = []
    | otherwise  = [char:CharListRead filewithchangedreadpointer]
where
    (readok,char,filewithchangedreadpointer) = sfreadc f
```

This completes the file copy function but we do not have a file copy *program* yet. What is missing is functions to open and close the files in question and, of course, we have to arrange that the file system is accessible. A copy function that also opens and closes files is given below:

```
CopyFile :: String String *Files -> *Files
CopyFile inputfname outputfname filesys
| readok && writeok && closeok = finalfilesystem
| not readok    = abort ("Cannot open input file: '"   +++ inputfname  +++ "'")
| not writeok   = abort ("Cannot open output file: '"  +++ outputfname +++ "'")
| not closeok   = abort ("Cannot close output file: '" +++ outputfname +++ "'")
where
    (readok,inputfile,touchedfilesys) = sfopen inputfname FReadText filesys
    (writeok,outputfile,nwfilesys) = fopen outputfname FWriteText touchedfilesys
    copiedfile                 = CharFileCopy inputfile outputfile
    (closeok,finalfilesystem)  = fclose copiedfile nwfilesys
```

The definition above uses the library functions `fopen` and `sfopen` to open files. The difference between them is that `fopen` requires the file to be unique and `sfopen` allows sharing of the file. Both functions have argument attributes indicating the way the file is used (`FReadText`, `FWriteText`). Another possible attribute would be `FAppendText`. Similar attributes exist for dealing with files with data.

Accessing the file system itself means accessing the 'outside world' of the program. This is made possible by allowing the `Start` rule to have an abstract parameter `World` which encapsulates the complete status of the machine.

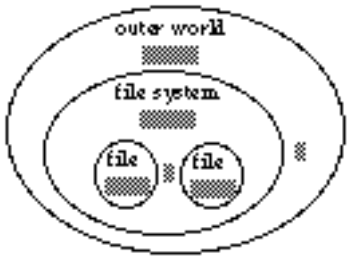

Figure 5.1 The abstract type `World` encapsulating the file system.

Employing unique environment passing, functions have been defined in the library that semantically produce new worlds fetching the file system from the world (`openfiles`) and putting it back in again (`closefiles`). The final result of the Clean program is then the world it delivers.

```
Start :: *World -> *World
Start world = CopyFileInWorld world

CopyFileInWorld :: *World -> *World
CopyFileInWorld world = newworld
where
    (filesystem,worldwithoutfilesystem) = openfiles world
    finalfilesystem     = CopyFile inputfilename outputfilename filesystem
    newworld            = closefiles finalfilesystem worldwithoutfilesystem
    inputfilename       = "source.txt"
    outputfilename      = "copy.txt"
```

This completes the file copy program.

Other ways to read files are line-by-line or megabyte-by-megabyte which may be more appropriate depending on the context. The corresponding read-functions are given below.

```
LineListRead :: File -> [String]
LineListRead f
    | sfend f   = []
    | otherwise = [line:LineListRead filerest]
where
    (line,filerest) = sfreadline f // line still includes newline character
```

```
MegStringsRead :: File ->[String]
MegStringsRead f
    | sfend f   = []
    | otherwise = [string:MegStringsRead filerest]
where
    (string,filerest)   = sfreads f MegaByte
    MegaByte            = 1024 * 1024
```

The functions given above are lazy. So, the relevant parts of a file are read only when this is needed for the evaluation of the program[1].

5.2 Combination of input/output functions

Consider the following definitions:

```
WriteAB :: *File -> *File
WriteAB file = fileAB
where
    fileA   = fwritec 'a' file
    fileAB  = fwritec 'b' fileA

WriteAB :: *File -> *File
WriteAB file = fwritec 'b' (fwritec 'a' file)
```

They are equivalent using slightly different styles of programming with environment passing functions.

A disadvantage of the first one is that new names have to be invented: `fileA` and `fileAB`. If such a style is used throughout a larger program one tends to come up with less clear names such as `file1` and `file2` (or even `file`` and `file``) which makes it harder to understand what is going on.

The second style avoids this but has as disadvantage that the order of reading the function composition is the reverse of the order in which the function applications will be executed.

Below some other styles of defining the same function are used (for this example one of the last two styles is preferable since they avoid both disadvantages mentioned above):

```
WriteAB :: (*File -> *File) //brackets indicate function is defined with arity 0
WriteAB = fwritec 'b' o fwritec 'a'

WriteAB :: (*File -> *File)
WriteAB = seq [fwritec 'a',fwritec 'b']

WriteAB :: *File -> *File
WriteAB file = seq [fwritec 'a',fwritec 'b'] file
```

With `seq` a list of state-transition function is applied consecutively. The function `seq` is a standard library function which is defined as follows:

```
seq :: [s->s] s -> s //restricted type: see section Polymorphic Uniqueness below
seq []     x = x
seq [f:fs] x = seq fs (f x)
```

[1] Sometimes it may be wanted to read a file completely before anything else is done. Below a strict read-function is given which reads in the entire file at once.

```
CharListReadEntireFile:: File -> [Char]
CharListReadEntireFile f
| not readok = []
| otherwise  = let! chars = CharListReadEntireFile filewithchangedreadpointer
               in   [char : chars]
where
    (readok,char,filewithchangedreadpointer) = sfreadc f
```

When functions are combined that perform both environment passing as well as produce other results similar style differences occur. When the types of these results are the same for all functions to be combined a simple variant of seq can be used:

```
:: St s a :== s -> (a,s)                // type of the functions to be combined
                                        // restricted type: see 5.2.3 below
seqList :: [St s a] s -> ([a],s)        // results are collected in a list
seqList [] state     = ([],state)
seqList [f:fs] state = ([a:as],state2)
where
    (as,state2) = seqList fs state1
    (a,state1)  = f state

readzipcode file = seqList [readarea,readregion] file
where
    readarea            = readstring 4
    readregion          = readstring 2
    readstring int file = freads file int
```

5.2.1 Monadic style

When a number of functions produce results of different type (e.g. freadc and freads) it is not possible to collect all these results in a list since elements of a list must have the same type. So, for that case another abstraction has to be used to grasp the essence of that kind of environment passing in general. The idea is that a function that takes the state and produces a tuple of the state and a result is combined by the function `bind` with a function that takes that result and produces another result-delivering state transition function. The result of `bind` is again such a result-delivering state-transition function. This way of combining function is called the *monadic* style of programming. (The type St s a together with the manipulation functions `bind` and return is called the *monad*.)

```
(`bind`) infix 0 :: (St s a) (a -> (St s b)) -> (St s b) // restricted type
(`bind`) f_sta a_fstb = stb
where
    stb st = a_fstb a nst
    where
        (a,nst) = f_sta st

return :: a -> (St s a)                                 // restricted type
return x = \s -> (x,s)

readzipcode :: (*File -> ((Int,Char,Char),*File))
readzipcode file
 =  freadint `bind` \(b1,i)->
    freadchar `bind` \(b2,c1)->
    freadchar `bind` \(b3,c2) ->
    if (b1 && b2 && b3) (return (i,c1,c2))
                        (abort "readzipcode failure")
where
    freadint  file = ((b,i),file1) where (b,i,file1) = freadi file
    freadchar file = ((b,c),file1) where (b,c,file1) = freadc file
```

The example[2] above shows how state-transition functions producing different types of results can be combined in this monadic style.

In languages where uniqueness types are not available enforcing the use of this monadic style of programming (via modular abstraction) is sufficient to guarantee referential transparency provided that the library functions have been proven to satisfy this property.

[2] In Clean 1.1 it is required to write the function in lambda notation between parentheses.

5.2.2 Nested scope style[3]

Another way to deal with a number of functions producing results of different type passing around an environment, is to make use of nested scopes of let before definitions (indicated by `let` or `#`). In that style the example above can be written as follows:

```
readzipcode :: *File -> ((Int,Char,Char),*File)
readzipcode file
# (b1,i,file)    = freadi file
# (b2,c1,file)   = freadc file
# (b3,c2,file)   = freadc file
| b1 && b2 && b3 = ((i,c1,c2), file)
| otherwise      = abort "readzipcode failure"
```

Due to the nesting of scopes of the let expressions the definition is equivalent to

```
readzipcode :: *File -> ((Int,Char,Char),*File)
readzipcode file
# (b1,i,file1)   = freadi file
# (b2,c1,file2)  = freadc file1
# (b3,c2,file3)  = freadc file2
| b1 && b2 && b3 = ((i,c1,c2), file3)
| otherwise      = abort "readzipcode failure"
```

The second style is not recommended since many different names have to be invented for the environment that is passed around. You have to invent all these names when you pass the environment in ordinary local definitions after a `where`.

The nested scope notation can be very nice and concise but, as is always the case with scopes, it can also be dangerous: the same name is used on different spots while the meaning of the name is not always the same (one has to take the scope into account which changes from definition to definition). However, the notation is rather safe when it is used to thread parameters of unique type. The type system will spot it (and reject it) when such parameters are not used in a correct single threaded manner. We certainly do not recommend the use of let before expressions to adopt a imperative programming style for other cases.

The scope of the variables introduces by the `#`-definitions is the part of the right-hand side of the function following the `#`-definition. The right-hand side `#`-definition and the `where`-definitions are excluded from this scope. The reason to exclude the right-hand of the `#`-definition is obvious from the example above. When the body of the `#`-definition is part of the scope the variable `file` would be a circular definition. The reason to exclude the `where`-definitions is somewhat trickier. The scope of the `where`-definitions is the entire right-hand side of the function alternative. This includes the `#`-definitions. This implies that when we use the variable `file` in a `where`-definition of `readzipcode` it should be the original function argument. This is counter intuitive, you expect `file` to be the result of the last `freadc`. When you need local definitions in the one of the body of such a function you should use `let` or `with`. See the language manual and chapter 6 for a more elaborate discussion of the various local definitions.

5.2.3 Polymorphic Uniqueness

In fact, the types of `seq`, `seqList`, `return` and `(`bind`)` in the library are more general. The type definitions contain polymorphic variables for the uniqueness attributes (see subsection 4.3). The types as given below are *inferred* by the compilers type inferencer. So, a programmer does not have to specify them.

```
seq              :: ![.(.s -> .s)] .s -> .s         // fn-1 (..(f1 (f0 x))..)
seqList          :: ![St .s .a] .s -> ([.a],.s)     // fn-1 (..(f1 (f0 x))..)
(`bind`) infix 0 :: u:(St .s .a) u:(.a -> .(St .s .b)) -> u:(St .s .b)
return           :: u:a -> u:(St .s u:a)
```

[3]Nested scope style makes use of let before definitions which are introduced in Clean version 1.2. So, this style cannot be used with earlier versions.

Furthermore, these types contain exclamation marks indicating that the function is strict in the corresponding argument.

5.3 Some Simple Dialogs

5.3.1 A File Copy Dialog

The previous sections showed us how to write a program that changes the filesystem. The file names however were coded directly in the program. Of course one would want to specify such parameters in an interactive way by using a dialog. For this purpose, it must not only be possible to fetch the filesystem out of the world but also the events that are generated by the user of the program (keys typed in, mouse clicks) have to be accessible and response on the screen must be given.

In Clean itself a library is written that makes it possible to address this event queue and deal with monitor output. Similar to addressing the filesystem, the event queue can be fetched as an abstract unique part from the world (using `OpenEvents` and `CloseEvents`).

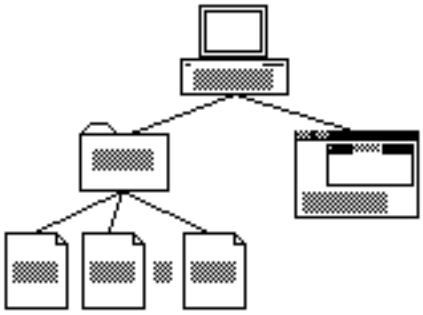

Figure 5.2 The world encapsulating the file system and the event queue.

To deal with events the programmer defines an algebraic data structure which specifies what kind of events are reacted upon and which event-handling function has to be applied when the event occurs. Each event-handling function has two unique parameters: the local state and the abstraction (called the `IOState`) of the states of the components (called devices) of the interaction.

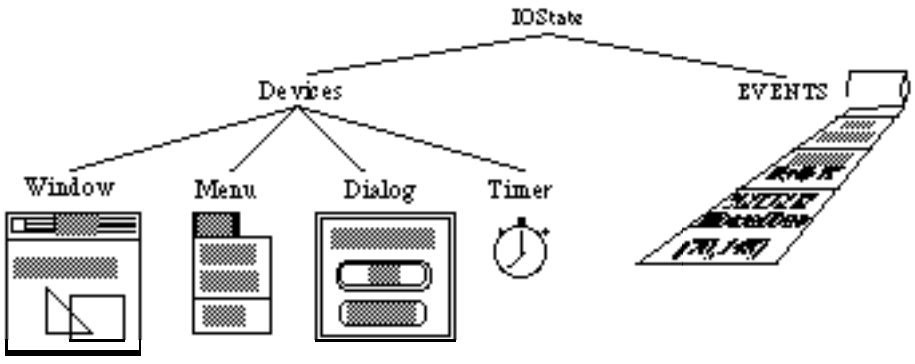

Figure 5.3 The `IOState` and its components.

This algebraic data structure defines the abstract devices of the program. It is given as a parameter to the library function `StartIO` which also takes an initial state, initial event-handling functions to perform and the event queue taken from the world. This function `StartIO` will deal with all interactions until the function `QuitIO` is called. It then delivers the final state and the list of leftover events.

The events handled by StartIO are all inputs of your program. Typical examples are pressing or releasing of keys at the keyboard, mouse clicks or mouse movements in a window, selection of items form the menu system, and timers. The events are handled in the order of occurence in the event stream. The function StartIO will search in the device definitions which call-back function (handler) is specified to handle this event. The call-back function found is applied to the current program state and IO-state. The IO-state contains all devices of the program. Using the appropriate functions these devices can be changed by the call-back functions. The program state is determined by the propgram. The initial program state is an argument of the application of StartIO.

All call-back functions delivers a tuple containing the new program state and new IO-state. The function StartIO will supply these states as arguments to the call-back function corresponding to the next event. This continues until one of the call-back functions applies QuitIO to the IO-state. The functions StartIO delivers a tuple containing the final program state and the unprocessed events after the evaluation of QuitIO. Events without an appropriate handler are ignored.

The function StartIO also has an argument containing

In the case of the file copy program the local state can consist of just the filesystem[4].

```
module copyfile

import StdEnv,deltaEventIO,deltaFont,deltaDialog,deltaSystem

Start :: *World -> *World
Start world = CopyFileDialogInWorld world

CopyFileDialogInWorld :: *World -> *World
CopyFileDialogInWorld world = newworld
where
    (filesystem,worldwithoutfilesystem) = openfiles world
    (events,emptyworld)         = OpenEvents worldwithoutfilesystem
    (finalstate,leftoverevents) = StartIO devicedefs initstate initialio events
    finalfilesystem             = finalstate
    worldwithevents             = CloseEvents leftoverevents emptyworld
    newworld                    = closefiles finalfilesystem worldwithevents

    devicedefs  = [ MenuSystem   [QuitMenu]
                  , DialogSystem [CopyFileDialog CopyFile]
                  ]
    initstate   = filesystem
    initialio   = []
```

The device definitions define the menu and the dialogue for this program. The Clean system takes care of drawing these menus and dialogues. The device definitions specify what will happen when an menu item is selected, or a dialogue button is pressed.

Apart from the last three local definitions of the function above this function could be used for just about any interactive program. In that case however, the state will usually not just be the filesystem but also certain program dependent values will be present in the state (e.g. the number of windows opened or the chosen settings of a game). So, for software engineering reasons it is the best choice to define the state as a record type containing the required values. This makes extension of the state easy since only the type and the initialisation has to be adapted since functions that use already existing field names do not have to be changed at all.

```
:: *ProgState   = {files :: Files}
```

[4]Do nopt forget to import the required modules when you write interactive programs. On Unix and Linux systems you should also link the appropriate libraries for window based programs. See the documentation of your Clean distribution.

```
CopyFileDialogInWorld :: *World -> *World
CopyFileDialogInWorld world = newworld
where
    (filesystem,worldwithoutfilesystem) = openfiles world
    (events,emptyworld)         = OpenEvents worldwithoutfilesystem
    (finalstate,leftoverevents) = StartIO devicedefs initstate initialio events
    finalfilesystem             = finalstate.files
    worldwithevents             = CloseEvents leftoverevents emptyworld
    newworld                    = closefiles finalfilesystem worldwithevents

    devicedefs  = [ MenuSystem   [QuitMenu]
                  , DialogSystem [CopyFileDialog CopyFile]
                  ]

    initstate   = {files = filesystem}
    initialio   = []
```

Programming for window systems requires some knowledge of the corresponding terminology (menu, pop-up menu, modal dialog, radio button, close box etc.). Such knowledge is assumed to be present with the reader (as a user of such systems). However, when it is felt appropriate, some of this terminology will be explained when it occurs in an example.

The definition of the `Quit` menu is just an algebraic data structure which is built dynamically using the constructors `PullDownMenu` and `MenuItem`. The definition of the type `MenuDef` in the library specifies the different options that are supported by the library.

```
QuitMenu :: MenuDef *s (IOState *s)
QuitMenu = PullDownMenu menId "File" Able
                                [MenuItem qId "Quit" (Key 'Q') Able QuitFun]
where
    QuitFun s io  = (s,QuitIO io)
    [menId,qId:_] = [0..]
```

Each part of the device definitions can be addressed by its Id (an integer identifying the part of the device). Using these Id's dialogs can be closed and opened, windows can be written, text typed in a dialog can be read, menu items can be disabled etcetera, etcetera. It is a recommended programming style to keep these Id's as much as possible local. The program above uses a local list definition with a wild-card which is easily extended when more Id's are required.

The dialogue of the copy file program is depicted in figure 5.4.

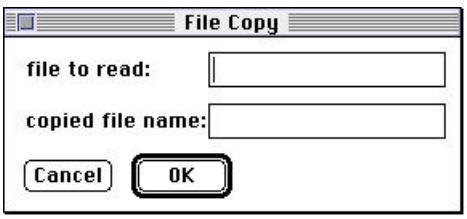

Figure 5.4 The dialog result of the function `CopyFileDialog`.

The appearance of this dialogue is determined by function `CopyFileDialog`. The function to be executed when the **ok** button is pressed, is passed as argument to `CopyFileDialog`. The dialogue definition consists of an enumeration of its components.

```
CopyFileDialog :: (String String *Files -> *Files)
                                    -> DialogDef *ProgState (IOState *ProgState)
CopyFileDialog copyfun
 =  CommandDialog dlgId "File Copy" [] okId
    [   StaticText srcId Left ("file to read: ")
    ,   EditText srcInputId (RightTo srcId) inputlength nrlines defaultinput
    ,   StaticText dstId Left ("copied file name: ")
    ,   EditText dstInputId (Below srcInputId) inputlength nrlines defaultinput
    ,   DialogButton cancelId (Below dstId) "Cancel" Able (cancel dlgId)
```

```
        ,   DialogButton okId (RightTo cancelId) "OK" Able (ok copyfun)
        ]
    where
        inputlength  = MM 50.0
        nrlines      = 1
        defaultinput = ""

        cancel id dlginfo s io = (s, CloseDialog id io)
        ok fun dlginfo s io    = ({s & files = newfilesys},CloseDialog dlgId io)
        where
            newfilesys      = fun inputfilename outputfilename s.files
            inputfilename   = GetEditText srcInputId dlginfo
            outputfilename  = GetEditText dstInputId dlginfo

        [dlgId,srcId,srcInputId,dstId,dstInputId,cancelId,okId:_] = [0..]
```

The dialog definition itself is again an algebraic data structure (of type `DialogDef *ProgState (IOState *ProgState)`). The data structure is built dynamically and interpreted by the library to create the proper reflections on the screen. Through the use of constructors and id's to indicate parts of the dialog, the dialog layout can be specified (`Below dstId`). The structure contains functions that are called when the corresponding button is selected. These functions can address the contents of the dialog with the library function `GetEditText`. The example above shows how currying can be very useful for such definitions (`ok copyfun` and `cancel dlgId`).

It is important to realise that the devices are specified by ordinary data structures and functions in Clean. The IO-system is an library, not a part of the language. This implies that the definition of the devices can be manipulated just like any other data structure in Clean. In this way all kinds of devices can be created dynamically.

The look and feel of dialogues and other devices is dependent of the operating system used. In this book we will usually show examples utilising an Apple Macintosh. The Clean code can be used unchanged on other platforms where Clean is available. The compiled program will have the appropriate look and feel for that platform.

A disadvantage of the dialog defined above is that it does not enable the user to browse through the file system to search for the files to be copied. Using the functions from the library module `deltaFileSelect` such dialogs are created in the way that is standard for the actual machine the program will be running on.

```
import deltaFileSelect

FileReadDialog fun state io
    | notcancel = fun name nwstate nwio
    | otherwise = (nwstate,nwio)
where
    nwstate                         = {state & files = nwfiles}
    (notcancel,name,nwfiles,nwio)   = SelectInputFile state.files io

FileWriteDialog fun state io
    | notcancel = fun name nwstate nwio
    | otherwise = (nwstate,nwio)
where
 nwstate                        = {state & files = nwfiles}
 (notcancel,name,nwfiles,nwio)  = SelectOutputFile prompt default state.files io
 prompt  = "Write output as:"
 default = "file.copy"
```

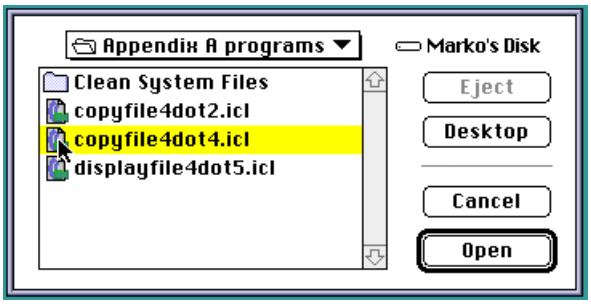

Figure 5.5 A standard `SelectInputFile` dialog.

An important advantage of the use of a library as described above is that the same program can run without any change on different machines while the library is such that on each machine the resulting menus, dialogs and windows adhere to the look and feel which is common on that machine.

Incorporation of these functions into a program is treated in one of the exercises.

5.3.2 A Function Test Dialog

Suppose you have written a function `GreatFun` and you want to test it with some input values. A way to do this is to use 'console' mode and introduce a start rule with as its right-hand-side a tuple or a list of applications of the function with the different input values.

```
Start = map GreatFun [1..1000]
```

or e.g.

```
Start = (GreatFun 'a', GreatFun 1, GreatFun "GreatFun")
```

Reality learns us that in this static way of testing less variety in testing occurs compared with dynamic interactive testing. For interactive testing, a dialog in which input-values can be typed, will be much helpful.

The previous section has shown how to define a dialog. Here we will define a function that takes a function as an argument and produces the IO-system with the dialog with which the function can be tested. We want this definition to be very general. We use overloading to require that the input values (typed in the dialog as a `String`) can be converted to the required argument of the test function.

The overloaded test dialog can be used to test a function on a structured argument (a list, a tree, a record, ...) straightforwardly. All that is needed is to write instances of `fromString` and `toString` for types or subtypes if they are not already available.

```
definition module funtest

from StdString import String
import StdEnv,deltaIO

FunctionTest :: (a -> b) a  .DialogTitle .TextWidth .Int *World
                                -> .World | toString b & fromString, toString a

/* Apply FunctionTest e.g as a test for the sin function on reals: 0.0 is the
initial value in the inputbox.

module functiontest
import StdEnv,IOUtilities

Start world = FunctionTest sin 0.0 "sin" (MM 100.0) 5 world

instance fromString Real where fromString s = toReal s
/* needed only if it is not defined already in some library */
*/
```

The corresponding implementation is a standard menu and dialog system that calls a function that defines the dialog. The program requires no local state, an empty list is given as parameter to `StartIO`. The program does nothing with files so there is no need to open the file system. The menu system is similar to the file copy program. A small difference is that the name of the open command is generated by the function `FunctionTest` using the parameter `fname`.

```
implementation module funtest
import StdEnv
import deltaDialog,deltaEventIO,deltaSystem
import deltaWindow,deltaControls,deltaFileSelect

FunctionTest :: (a -> b) a .DialogTitle .TextWidth .Int *World
                                  -> .World | toString b & fromString, toString a
FunctionTest function initval fname width nrlines world
= CloseEvents leftoverevents worldwithoutevents
where
    (events, worldwithoutevents) = OpenEvents world
    (_, leftoverevents)          = StartIO devicedefs [] [] events

    devicedefs= [ DialogSystem [FunDialog]
                , MenuSystem [PullDownMenu DontCareId "File" Able menuitems]
                ]

    menuitems = [ MenuItem DontCareId ("Open "+++fname) (Key 'O') Able Open
                , MenuSeparator
                , MenuItem DontCareId "Quit"            (Key 'Q') Able Quit
                ]

    FunDialog = functiondialog fname width nrlines dialogfunction dialoginitval
    where
        dialoginitval    = toString initval
        dialogfunction x = toString (function (fromString x))

    Open s io = (s,OpenDialog FunctionDialog io)
    Quit s io = (s, QuitIO io)

DontCareId :== 0
```

The function that defines the dialog itself (`functiondialog`) is defined below and used above with `String` parameters and with a `String -> String` function. Using `toString` and `fromString` combined with the general polymorphic function parameter (`function`) a function from string to string is created (`dialogfunction`) that is passed to the function `functiondialog`.

The definition of the function `functiondialog` itself is given below. It is quite similar to the file copy dialog. Note that the cancel function can be used in general for many kinds of dialogs.

```
functiondialog :: .DialogTitle .TextWidth .Int (String -> String) String
                                                  ->  .DialogDef *a (IOState *a)
functiondialog name width nrlines fun initstring
= CommandDialog dlgId name [] okId
        [ StaticText intextId Left (name+++" input: ")
        , EditText inputId (RightTo intextId) width nrlines initstring
        , StaticText outtextId Left (name+++" output: ")
        , EditText outputId (Below inputId) width nrlines ""
        , DialogButton cancelId Left "Cancel" Able (cancel dlgId)
        , DialogButton okId (Below outputId) "OK" Able (ok fun)]
where
    ok fun dlginfo s io
    = (s,ChangeDialog dlgId [ChangeEditText outputId (fun input)] io)
    where
        input = GetEditText inputId dlginfo

    [dlgId,intextId,outtextId,inputId,outputId,cancelId,okId:_]
    = [inc DontCareId..]
```

```
cancel :: .DialogId .DialogInfo *s (IOState *s) -> (*s,IOState *s)
        // type is more restrictive than deduced type
cancel id dialoginfo s io = (s, CloseDialog id io)
```

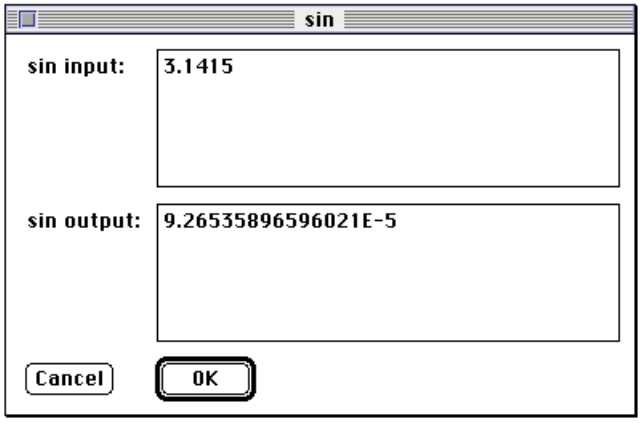

Figure 5.6 An example of the use of the `FunctionTest` dialog system generator.

This completes already the full definition of a general dialog for testing polymorhic functions with one parameter. For another number of parameters the dialog is easily adapted. Writing a testing dialog for a function with any number of parameters is harder. One would have to assume that the arguments are collected in some kind of structure since the language has no facilities to distinguish at run-time between a function type result (requiring an extension of the input fields) and a non-function type result. Writing a test dialog for such a structure of arguments (a list, a tree, a record, ...) is straightforward again (in fact: all that is needed is to write instances of `fromString` and `toString`).

5.3.3 An Input Dialog for a Menu Function

Similarly, an input dialog for a menu function can be defined. In this case we have overloaded the dialog definition itself by requiring `fromString` to be defined on the result of `GetEditText`.

```
inputdialog :: .DialogTitle .TextWidth
                  (a -> .(MenuFunction *s (IOState *s))) *s (IOState *s)
                                              -> (*s,IOState *s) | fromString a
inputdialog name width fun s io = (s,OpenDialog dialogdef io)
where
    dialogdef
    = CommandDialog dlgId name [] okId
        [StaticText nameId Left (name+++": ")
        ,EditText inputId (RightTo nameId) width 1 ""
        ,DialogButton cancelId (Below inputId) "Cancel" Able (cancel dlgId)
        ,DialogButton okId (RightTo cancelId) "OK" Able (ok fun)]

    ok fun dlginfo s io = fun input s (CloseDialog dlgId io)
    where
        input = fromString (GetEditText inputId dlginfo)

    [dlgId,nameId,inputId,cancelId,okId:_] = [0..]
```

The type `MenuFunction` used in the type of `inputdialog` is defined in the library.

```
::  MenuFunction *s *io :== s -> *(io -> (s, io))
```

It is the type of a menu function like `open`, `test sin` and `quit` in the previous examples.

This input dialog can be used for all kinds of 'menu' functions (i.e. all functions with the right type operating on the outermost level: not only the functions that are present in the menus) that require a single (structured) input. The result of applying the function `inputdialog` to a name, a width and a menu function is again a menu function incorporating the extra input!

5.3.4 More Generic Dialog Definitions

In this way a collection of general definitions can be made that define most of the look and feel of an entire program. A few more examples are given below. They are self-explanatory.

```
/*warning on function to be applied: default Cancel */
warnCancel :: a .(MenuFunction *s (IOState *s)) *s (IOState *s)
                                                -> (*s,IOState *s) | toString a
warnCancel info fun s io
| choiceId == cancelId = (s,nio)
| otherwise            = fun s nio
where
    (choiceId,nio)    = OpenNotice warningdef io
    warningdef        = Notice [toString info] (NoticeButton cancelId "Cancel")
                                                          [NoticeButton okId "OK"]
    [cancelId,okId:_] = [0..]

/*warning on function to be applied: default OK */
warnOK :: a .(MenuFunction *s (IOState *s)) *s (IOState *s)
                                                -> (*s,IOState *s) | toString a
warnOK info fun s io
| choiceId == cancelId = (s,nio)
| otherwise            = fun s nio
where
    (choiceId,nio)    = OpenNotice warningdef io
    warningdef        = Notice [toString info] (NoticeButton okId "OK")
                                                [NoticeButton cancelId "Cancel"]
    [cancelId,okId:_] = [0..]

/*message to user: continu on OK */
inform  :: [String] (IOState *s) -> IOState *s
inform strings io = nwio
where
    (_,nwio) = OpenNotice (Notice strings (NoticeButton DontCareId "OK") []) io
```

The functions above can be used to inform and warn the user of the program but also to supply information to the programmer about arguments and (sub)structures when a specific function is called. The latter can be very helpful when debugging the program.

5.4 A simple window

Programming windows is more elaborate than programming a dialog (a dialog has more structure so the library can deal with most of the work). Consequently, a window must have an *update function* that redraws (part of) the window when required (e.g. when the window is put in front of another window or when it is scrolled). Furthermore, a window usually has a *keyboard function* and a *mouse function* to deal with characters typed in and with mouse actions.

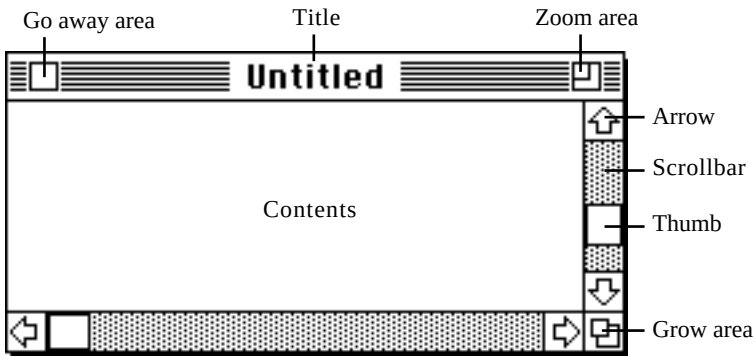

Figure 5.7 Some window terminology.

The contents of the window is composed of pixels. A pixel is a single point of the drawing area. Each pixel has an unique position: a `Point`. A `Point` is a pair integers. The origin, usually the point `(0,0)`, is the left upper corner of the drawing. As a matter of fact it is possible to use coordinates different from `(0,0)` for the left upper corner. The coordinates increase to the right and down.

The window scrolls over this picture area. The Clean system takes care of scrolling zooming and growing of the window. The actions associated mouse events, keyboard events and with clicking in the go away area are determined by the program. You program always works in the coordinate system of the picture. When your program draws something in the part of the picture area that is not in the current window nothing happens. In order to speed up drawing you can define the drawing functions such that only items inside the current window are shown. This is only worthwhile when drawing happens to be (too) time consuming.

The parts of the window that are currently outside the window, or are hidden beyond some other window, are not remembered by the system. In order to restore these parts of the picture on the screen the window is equipped with an update function. This update function has as argument the list of rectangles to be updated, the program state and the io state. For simple drawings it can be appropriate to redraw the entire picture as window update. More complex drawings require a more sophisticated update function in order to avoid time in drawing objects outside the window. This is illustrated by the function `updatefunction` in the following example.

Let us take the functions to read in a file and make a program that shows the contents of the file in a window extended with the possibility of selecting (highlighting) a line with the mouse and with the possibility of scrolling using keyboard arrows. This will give us a simple program with a window definition with an update function, a keyboard function and a mouse function.

The overall menu structure is straightforward. The state type is extended with two fields to indicate whether a line is selected and which line is selected.

```
module displayfileinwindow
import StdEnv, LibExt

:: *ProgState = {   select          :: Bool
                ,   selectedline    :: Int
                ,   files           :: Files
                }

Start world = closefiles finalfs (CloseEvents finales world``)
where
    ({files=finalfs},finales)   = StartIO [MenuSystem Menus] initstate [] es
    (es,world`)                 = OpenEvents world
    (fs,world``)                = openfiles world`

    initstate = {select=False,selectedline=abort "No line selected",files=fs}

    Menus     = [   PullDownMenu DontCareId "File" Able
                    [   MenuItem DontCareId "Read File..." (Key 'O') Able
                            (FileReadDialog (Show LineListRead))
                    ,   MenuSeparator
                    ,   MenuItem DontCareId "Quit" (Key 'Q') Able Quit
                    ]
                ]

    Quit s io = (s, QuitIO io)
```

The function `Show` takes a file access function, opens the file, puts the new files in the state and calls `DisplayInwindow` to display the result in a window.

```
Show readfun name s io
    | not readok = abort ("Could not open input file '" +++ name +++ "'")
    | otherwise  = ({s & files = nwfiles}, DisplayInWindow (readfun file) io)
where
    (readok,file,nwfiles) = sfopen name FReadText s.files
```

The window definition specifies the usual attributes and passes the text (a list of strings) to the update function as a list of lines (each represented as a list of characters).

```
DisplayInWindow text io = OpenWindows [windowdef] io
where
    windowdef
        = ScrollWindow DontCareId (0,0) "Read Result"   // id,position,title
         (ScrollBar (Thumb (~whiteMargin)) (Scroll Font.width))  // horizontal
         (ScrollBar (Thumb 0)              (Scroll Font.height)) // vertical
         ((~whiteMargin,0),(maxLineWidth,length lines*Font.height)) // domain
         (10,10) (640,480)                      // minimum size,init size
         (updatefunction lines)                 // window drawfunction
         [  Keyboard Able getkeys               // keyboard handling
         ,  Mouse Able getmouse                 // mouse handling
         ]
    where
        lines = splitby '\n' (flatten (map fromString text))

    whiteMargin  = 5
    maxLineWidth = 1024
```

The units of scrolling and the size of the domain are defined using the font sizes which are taken from the default font of the application. This font information is defined as a *global graph* (i.e. evaluated at most once) for the program.

```
:: InfoFont =   {   font    :: Font
                ,   width   :: Int
                ,   height  :: Int
                ,   up      :: Int
                }
Font        =:  {   font    = fnt
                ,   width   = maxwidth
                ,   height  = ascent+descent+leading
                ,   up      = ascent+leading
                }
where
    (ascent,descent,maxwidth,leading)   = FontMetrics fnt
    (_,fnt)                             = SelectFont name styles size
    (name,styles,size)                  = DefaultFont
```

The update function of a window is called automatically when (part of) the window must be redrawn. It has the list of domains that must be redrawn as a parameter. Furthermore it has the state as a parameter. Its result is a tuple of the state and a list of draw functions that are to be applied on the window.

In this case the update function is defined locally within the definition of the window to be able to use the defined constants `whiteMargin` and `maxLineWidth` directly and not as parameters. In order to keep things relatively simple the complete lines are drawn even when part of them is outside the redraw area (this has no visible effect apart from a very small inefficiency).

```
updatefunction textlines domains s=:{select,selectedline}
    =   (s,flatten (map update domains))
where
    update domain=:((_,top),(_,bot)) // draw (again) in between top and bot
     =  [   EraseRectangle ((~whiteMargin,top),(maxLineWidth,bot))
        :   drawlines (tolinenumber top) (tolinenumber (dec bot)) textlines
        ]

    drawlines first last textlines
     =  hilite ++
        [   MovePenTo (0,(towindowcoordinate first) + Font.up)
        :   flatten (map drawline (textlines%(first,last)))
        ]
```

```
        where
            hilite
                |   select
                &&  (selectedline >= first || selectedline <= last)
                            = hiliteline selectedline
                | otherwise = []

        drawline xs
         =  [   DrawString line
            ,   MovePen (~(FontStringWidth line Font.font),Font.height)
            ]
        where
            line = toString xs
```

The drawing functions from the library use, of course, window co-ordinates in the window domain while each program usually has its own co-ordinates in (a part of) its state. So, a program will usually contain transformation functions between the different sets of co-ordinates.

In this case the program will have to transform window co-ordinates to line numbers and vice versa.

```
    tolinenumber windowcoordinate = windowcoordinate / Font.height

    towindowcoordinate linenumber = linenumber * Font.height // top of the line
```

Using these transformations it is simple to write a function that highlights a line.

```
    hiliteline linenr   =   [   SetPenMode HiliteMode
                            ,   FillRectangle (towindowrectangle linenr)
                            ,   SetPenNormal
                            ]

    towindowrectangle linenumber
                = ((~whiteMargin,winco), (maxLineWidth,winco + Font.height))
    where
        winco   = towindowcoordinate linenumber
```

The keyboard function is called automatically when a key is hit. It has the keyboard information (is it a keydown?, which key?, with maybe an extra so-called *meta-key* or *modifier* such as shift, alt/option, command or control down?) as a parameter and of course the `state` and the `iostate`. Its result is a tuple of `state` and `iostate`.

In this case the action performed is calling `ChangeActiveScrollBar` which in its turn will cause a call of the window update function again.

```
    getkeys (kcode,kstate,modifs=:(shift,opt,comm,contr)) s io
        | IsKeyUp kstate    = (s,io)
        | key == LeftKey    = horscroll (horthumb - Font.width)  s io1
        | key == RightKey   = horscroll (horthumb + Font.width)  s io1
        | key == UpKey      = verscroll (verthumb - Font.height) s io1
        | key == DownKey    = verscroll (verthumb + Font.height) s io1
        | key == PgUpKey    = verscroll (verthumb - pagesize)    s io1
        | key == PgDownKey  = verscroll (verthumb + pagesize)    s io1
        | otherwise         = (s,io)
    where
        key                 = toChar kcode
        pagesize            = verdown - verthumb
        (((horthumb,verthumb),(_,verdown)),io1)
                            = ActiveWindowGetFrame io
        horscroll newvalue  = ChangeActiveScrollBar (ChangeHThumb newvalue)
        verscroll newvalue  = ChangeActiveScrollBar (ChangeVThumb newvalue)

        IsKeyUp KeyUp       = True
        IsKeyUp keystate    = False
```

The mouse function is called when a mouse action is performed. It has as its parameter the mouse information (position, no/single/double/triple/long click, modifier keys down) and of course the `state` and the `iostate`. Its result is a tuple of `state` and `iostate`.

In this case the mouse function changes the selected line, highlights the new selection and de-highlights the old one (by highlighting it again).

```
getmouse ((_,y),ButtonDoubleDown,_) s=:{select,selectedline=oldselection} io
    =   (   {s & select = True,selectedline = selection}
        ,   DrawInActiveWindow (changeselection oldselection selection) io
        )
where
    selection       = tolinenumber y

    changeselection old new
        | select    = hiliteline old ++ hiliteline new
        | otherwise = hiliteline new

getmouse _ s io = (s,io)
```

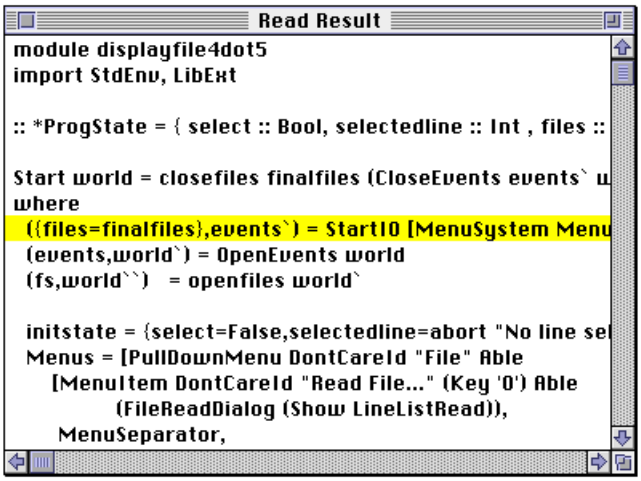

Figure 5.8 A view of the display file program when it has read in its own source.

5.5 Timers

Apart from reacting on user events by defining dialog systems or window systems as devices it is also possible to define timer devices with call-back functions (on state and io-state) that are called for timer events that are created by the system when a specified time has passed.

This can be used to show information on a regular basis or to change it e.g. in a shoot-them-up game. Another way of using a timer is to create some kind of background behaviour such as an autosave facility in an editor wich saves the edited file on a regular basis.

Adding a timer system is very similar to adding a dialog system: a timer device is added in the list of devices that is given to `StartIO`.

```
… StartIO [MenuSystem Menus,TimerSystem Timers] initstate [] events …
```

Such a timer device can contain a number of timers each with their own Id, interval and call-back function to be executed when the timer event occurs. In order to enable or disable a timer its Id must be known. Below a definition is given of a single timer which saves the displayed file in a copy every five minutes.

```
Timers = [Timer TimerId Unable TimerInterval Timerfunction]
where
    TimerInterval = 300 * TicksPerSecond // 300 seconds = 5 minutes

TimerId = 1

Timerfunction nrofintervalspassed state=:{name,lines} io
        = ({state & files = finalfiles},io)
where                                                   // nochecks on failure
    (_,finalfiles)      = fclose newfile nwfiles
```

```
        newfile             = LineListWrite lines file
        (_,file,nwfiles)    = fopen (name+++".copy") FWriteText state.files
```

Such a timer could be used for an autosave function that toggles the menu item title and function changing it from enabling to disabling and vice-versa.

```
        …
        MenuItem AutoSaveId "Enable AutoSave" (Key 'S') Able AutoSave,
        …

    AutoSaveId = 1

    AutoSave s io
        = (s,seq [  EnableTimer TimerId
                 ,  ChangeMenuItemFunctions [(AutoSaveId,NoAutoSave)]
                 ,  ChangeMenuItemTitles    [(AutoSaveId,"Disable AutoSave")]
                 ]  io)

    NoAutoSave s io
        = (s,seq [  ChangeMenuItemTitles [(AutoSaveId,"Enable AutoSave")]
                 ,  ChangeMenuItemFunctions [(AutoSaveId,AutoSave)]
                 ,  DisableTimer TimerId
                 ]  io)
```

Note that many of the program changes required for exercise 5.5 are also required for this auto-save function.

5.6 A line drawing program

In order to show how all pieces introduced above fit together we will show a complete window based program. The program is a simple line drawing tool. It is not intended as a complete drawing program, but to illustrate the structure of such programs. To limit the size of the program we have restricted ourselves to the basic possibilities. As a consequence there are a lot of desirable possibilities of a drawing program that are missing. Adding these features does not requires new techniques.

On the side of devices handled, the program is rather complete. It contains, of course, a window to make drawings. It uses the mouse to create and change lines. The drawing can be stored and retrieved from a file. There is a timer to remind the user to save the picture. Dialogues are used for a help function and the standard about dialogue. Finally there is a handler for input from the keyboard.

The program is called `Linedraw`. It starts by importing a long list of needed modules. The name of the modules indicates their function. These modules contain the type definitions and functions used to manipulate the io-system and all devices. You are encouraged to read the .dcl files whenever appropriate. These files determines the allowed constructs and contain useful comments about the used and semantics of these constructs.

```
module Linedraw

import StdEnv, deltaFileSelect, deltaDialog, deltaSystem, deltaTimer
import deltaEventIO, deltaIOSystem, deltaPicture, deltaWindow, deltaMenu
```

Since this programs handles drawing inside a window we will use window co-ordinates. The origin is the left upper corner and a point is indicated by a pair of integers. We use the following type synonyms from the module `Picture`. The type `Picture` itself is an unique abstract data type.

```
::  Point           :== (!Int, !Int);
::  Line            :== (!Point, !Point);

::  DrawFunction    :== Picture -> Picture;
```

As a first step the important data types of the program are defined. The type synonym `IO` is used as a handy name to indicate the unique state of the IO system. The state of the program is a record to enable extensions. The state contains the lines of the drawing. Since this is a pure line drawing tool, a list of lines is all there is for the drawing. The

file system is needed to store and retrieve drawings from file. Finally, the program state contains the name of the last file used, this is the default when we save the drawing again.

```
::  * IO    :== IOState ProgState
::  * ProgState =   {   lines   :: [Line]   // The drawing
                    ,   fsys    :: *Files   // The file system
                    ,   fname   :: String   // Name of file to store drawing
                    }
```

We will describe the program top down. This implies that we begin with the `Start` rule. The first part of this rule is fairly standard. The only difference with the earlier examples is that we use `#`-notation instead of local definitions after the `where` for environment passing. The initial program state, `InitState`, is define using a `with`, since it contains the file system, `fsys`, which is out of scope in the `where` definitions. The definition of the menus system and the window is completely standard. We discuss the dialogue and the timer below.

```
Start :: * World    ->  * World
Start world
 # (events, world) = OpenEvents world
 # (fsys,world)    = openfiles world
 # (s, events)     = StartIO [menus, window, dialog, timer] InitState [] events
    with InitState = {lines = [], fsys = fsys, fname = ""}
 = CloseEvents events (closefiles s.fsys world)
where
    menus   = MenuSystem [file,edit]
    file    = PullDownMenu DoNotCareId "File" Able
                [   MenuItem DoNotCareId "Open" (Key 'O') Able Open
                ,   MenuItem DoNotCareId "Save" (Key 'S') Able Save
                ,   MenuSeparator
                ,   MenuItem DoNotCareId "Quit" (Key 'Q') Able Quit
                ]
    edit    = PullDownMenu DoNotCareId "Edit" Able
                [   MenuItem DoNotCareId "Remove Line" (Key 'R') Able Remove
                ,   MenuSeparator
                ,   MenuItem DoNotCareId "Help"        (Key 'H') Able Help
                ]
    window  = WindowSystem
                [ ScrollWindow
                    WId (0,0) "Picture"                 // Window Id, Pos, Title
                    (ScrollBar (Thumb 0) (Scroll 10))   // Horizontal scroll bar
                    (ScrollBar (Thumb 0) (Scroll 10))   // Vertical scroll bar
                    PictDomain                          // Picture domain
                    MinWindowSize                       // Minimum window size
                    InitWindowSize                      // Initial window size
                    (\_ ps=:{lines} -> (ps,draw lines)) // Window update
                    [ Mouse Able MouseWait              // Window attribute list
                    , GoAway Quit
                    , Keyboard Able HandleKey
                    ]
                ]
    dialog  = DialogSystem
                [ AboutDialog "Linedraw" ((0, 0), (160, 25))
                    [MovePenTo (10, 10), DrawString "A line drawing tool"]
                    (AboutHelp "Help" Help)
                ]
    timer   = TimerSystem [Timer TId Unable time remaindSave]
```

During the development of such a program you can begin with a less elaborated user interface. In fact during the development of this example we started without dialogue system and timers.

It is convenient to construct programs in an incremental way. We begin with a very simple version of the program and add extensions one by one. The program is compiled and tested before each subsequent addition. For window based programs we can omit a part of the menu structure or use a no-operation for the menu functions: `\ s io -> (s,io)`. Also the window update function can be a no-operation in the first approximations of your

program. The mouse handler and keyboard handler of the window can be omitted in the beginning, or we can again use a no-operation.

The definition of the `Start` rule above contains a special purpose about dialogue. This dialogue is intended to give some information about the program. On the Macintosh it will become the first item in the -menu. Its definition differs slightly from the dialogues we have seen above. In fact the about dialogue is treated almost as an ordinary simple window. The difference with a window is that the Clean system will generate an **OK**-button and, if you indicate so, an **Help**-button. The structure of the dialogue definition and the allowed constructs are determined by the type definition in `deltaDialog`. The specified about dialogue looks like:

Figure 5.9 The about dialogue of the line drawing tool.

Further more the current `Start` rule determines that there is a timer. This timer will be used to remind the user of the drawing tool to save his work. Initially the timer is disabled. It will be enabled as soon as the something is changed to the drawing.

The first menu function that we implement is the function `Quit`. This enables us to leave the test versions of our program in a descent way. Fortunately the implementation is very simple, we simply quit the io system. This causes that the function `StartIO` yield the tuple `(s,events)`. The `Start` rule will close the events and the file system. The program is finished when this is done.

```
Quit :: ProgState IO -> (ProgState,IO)
Quit state io = (state, QuitIO io)
```

Next we went immediately to the difficult part of the program: mouse handling. This is generally a good strategy: do the difficult and interesting parts of the program first. The simple details that makes your program complete can be added later. The difficult part determines most likely the success of the program under construction. There is no excuse to spent time on simple work on a program that still risks to be changed fundamentally.

The first thing that needs to be done with the mouse is drawing lines. A line starts at the point where the mouse button is pushed and ends where the mouse button is released. While the mouse button is pushed the line under construction is drawn like a rubber band.

The mouse state which is argument of the mouse handler contains the current position of the mouse. We need to remember the start point of the line. The previous end point of the line is needed in order to erase the version of the line drawn. These points can be stored in the program state. We have chosen to pass them as arguments to a new mouse handler function. A more sophisticated version of the function `MouseWait` will be introduced very soon. We will need similar changes of the IO-state later, so we use the function `startDraw` to perform the desired changes of the IO-state. `startDraw` first disables the timer since we does not want the question whether the user wants that the save remainders interferes with drawing of a line. Next the pen is switched to `XorMode`. This is convenient for rubber band drawing. Drawing some object twice in `XorMode` restores the original picture. The initial version of the line is drawn, and the appropriate mouse handler is installed.

```
MouseWait :: MouseState ProgState IO -> (ProgState, IO)
MouseWait (pos,ButtonDown,_) state io = (state,startDraw [] pos pos io)
MouseWait _ state io = (state,io)

startDraw :: [DrawFunction] Point Point IO -> IO
startDraw fs s e io
```

```
 =  ChangeMouseFunction WId (MouseDraw s e)
    ((DrawInWindow WId (fs++[SetPenMode XorMode,DrawLine (s,e)])) (timerOff io))
```

The function `MouseDraw` takes care of drawing a line. If the mouse event is a `ButtonUp` the line is completed. The line is added to the state, the temporarily line is removed by drawing it again, the final line is drawn with the normal pen and the timer is enabled. For other mouse events we check whether the mouse is moved (`y == z`). If the mouse is not moved we are done. If it is moved we erase the old line by drawing it again, and draw the newline. The pen remains is `XorMode`.

```
MouseDraw :: Point Point MouseState ProgState IO -> (ProgState, IO)
MouseDraw x y (z,ButtonUp,_) state=:{lines} io
 = ({state & lines = [(x,z):lines]}
   ,timerOn (ChangeActiveMouseFunction MouseWait
    (DrawInActiveWindow [DrawLine (x,y),SetPenNormal,DrawLine (x,z)] io))
   )
MouseDraw x y (z,_,_) state io
 | y == z = (state,io)
          = (state
            ,ChangeActiveMouseFunction (MouseDraw x z)
                (DrawInActiveWindow [DrawLine (x,y),DrawLine (x,z)] io)
            )
```

With these functions you can compile your program again and draw some lines. You will soon discover that it is desirable to change the drawing. A very simple way to change the picture is by removing the last line drawn. This is accomplished by the menu function `Remove`. If there are lines overlapping with the line to be removed, it is not sufficient to erase that line. This would create holes in the overlapping lines. We simply erase the entire picture and draw all remaining lines again. With a some more programming effort the amount of drawing can be reduced, but there is currently no reason to spent this effort. Removing a line changes the picture so we make sure the timer is switched on. If the list of lines is empty, there is no line to be removed. We make the computer beep in order to indicate this error.

```
Remove :: ProgState IO -> (ProgState,IO)
Remove state=:{lines} io
 | isEmpty lines = (state,Beep io)
                 = ({state & lines = lines`}
                   , timerOn (DrawInWindow WId (draw lines`) io))
where lines` = tl lines

draw :: [Line] -> [DrawFunction]
draw ls = [EraseRectangle PictDomain,SetPenNormal:[DrawLine l \\ l <- ls]]
```

An other way to change the picture is by editing an existing line. If the user presses the mouse button with the shift key down very close to one of the ends of the line, that line can be changed. We use very close to the end of a line instead of at the end of a line since it appears to be difficult to position the mouse exactly at the end of the line.

We change the function `MouseWait`. First we check whether the shift key is pressed. If it is, we try to find a line end touched by the mouse. If such a line is found, we remove it from the state, and start drawing the line with the previous version as initial version. If no line is touched the program ignores this mouse event. If the shift key is not pressed, we proceed as in the previous version of the function `MouseWait`.

The function `touch` determines whether or not a point is very close to the end of a one of the given lines. Instead of yielding a Boolean, this function uses the type `Option`. In case of success the type `Option` also holds the value of the success. Here the success is the line touched and the list of all other lines. The type `Option` can be used often.

```
MouseWait :: MouseState ProgState IO -> (ProgState, IO)
MouseWait (pos,ButtonDown,(shift,_,_,_)) state=:{lines} io
 | shift = case touch pos lines of
           Yes ((s,e),ls) -> ({state & lines = ls}, startDraw (draw ls) s e io)
           No             -> (state, io)
```

```
        = (state,startDraw [] pos pos io)
MouseWait _ state io = (state,io)

:: Option x = Yes x | No

touch :: Point [Line] -> Option (Line,[Line])
touch p [] = No
touch p [l=:(s,e):r]
    | closeTo p s = Yes ((e,s),r)
    | closeTo p e = Yes (l,r)
                  = case touch p r of
                     Yes (t,x) -> Yes (t,[l:x])
                     no        -> no
where closeTo (a,b) (x,y) = (a-x)^2 + (b-y)^2 <= 10
```

Next we want to be able to store the drawing in a file and read it back. Each line is represented by its end points. Each of these points consists of two integers. A line is stored as four integers in a data file. We use the dialogue from `deltaFileSelect` to determine the name of the output file.

```
Save :: ProgState IO -> (ProgState,IO)
Save ps=:{fsys,fname} io
 # (ok,fn,fsys,io) = SelectOutputFile "Save as" fname fsys (timerOff io)
 | not ok          = ({ps & fsys = fsys},timerOn io)
 # (ok,file,fsys)  = fopen fn FWriteData fsys
 | not ok          = ({ps&fsys=fsys},inform ["Cannot open file"] (timerOn io))
 # file            = seq (flatten [map fwritei [a,b,x,y]
                                   \\ ((a,b),(x,y)) <- ps.lines ]) file
 # (ok,fsys)       = fclose file fsys
 | not ok          = ({ps&fsys=fsys},inform ["Cannot close file"] (timerOn io))
 = ({ps & fsys = fsys, fname = fn},resetTimer io)
```

We used the following function to generate a simple dialogue, called notice, when there is something wrong with opening or closing the file.

```
inform :: [String] (IOState *s) -> IOState *s
inform m io = snd (OpenNotice (Notice m (NoticeButton DoNotCareId "OK") []) io)
```

Opening and reading lines from a file is very similar. We open the file as a unique file in order to allow reuse. In this way the user can read a drawing from a file, change it and save it again in the same file. Each sequence of four integers found in the file is interpreted as a line. We do not do any checks on the format of the file. This implies that almost any data file can be interpreted as a drawing. For such a simple program this is sufficient.

```
Open :: ProgState IO -> (ProgState,IO)
Open ps=:{fsys} io
 # (ok,fn,fsys,io) = SelectInputFile fsys (timerOff io)
 | not ok          = ({ps & fsys = fsys},timerOn io)
 # (ok,file,fsys)  = fopen fn FReadData fsys // *File to allow reuse
 | not ok          = ({ps&fsys=fsys},inform ["Cannot open file"] (timerOn io))
 # (ints,file)     = readInts file
 # (ok,fsys)       = fclose file fsys
 | not ok          = ({ps&fsys=fsys},inform ["Cannot close file"] (timerOn io))
 =  ({ps & fsys = fsys, fname = fn, lines = lines}
    ,DrawInWindow WId (draw lines) (resetTimer io)
    )
    with lines = toLines ints

toLines :: [Int] -> [Line]
toLines [a,b,x,y:r] = [((a,b),(x,y)):toLines r]
toLines _           = []

readInts :: *File -> ([Int],*File)
readInts file
    # (end,file)    = fend file
    | end           = ([],file)
    # (ok,i,file)   = freadi file
    | not ok        = ([],file)
```

```
    # (is,file)     = readInts file
    = ([i:is],file)
```

As a next step we add the keyboard handler to the window of our drawing program. The arrow keys scroll the window and the back space key is equivalent to the menu item `remove`. `KeyUp` events are ignored by the first alternative, this implies that the handler reacts on `KeyDown` and `KeyStillDown` events. In order to do this we first collect some information of the window using `ActiveWindowGetFrame`. Then we do a case analysis of the key. Scrolling is done using the functions from `deltaWindow`. If the key is not recognised the computer will `Beep` as a slight error indicator.

```
HandleKey (_,KeyUp,_) s io = (s,io)
HandleKey (kcode,_,_) s=:{lines} io
 # (((htumb,vtumb),_),io) = ActiveWindowGetFrame io
 = case toChar kcode of
    LeftKey   -> ChangeActiveScrollBar (ChangeHThumb (htumb-10)) s io
    RightKey  -> ChangeActiveScrollBar (ChangeHThumb (htumb+10)) s io
    UpKey     -> ChangeActiveScrollBar (ChangeVThumb (vtumb-10)) s io
    DownKey   -> ChangeActiveScrollBar (ChangeVThumb (vtumb+10)) s io
    BackSpKey -> Remove s io
    _         -> (s, Beep io)
```

The function `Help` just shows a notice containing a few lines of text containing some information about using this program.

```
Help :: ProgState IO -> (ProgState,IO)
Help s io = (s,inform helptext io)

helptext :== ["Hold down shift to move end points of line"
             ,"Arrow keys scroll window"
             ,"Backspace deletes last line"
             ]
```

The last devices added is the times. After a predefined of time after the first change of the drawing a notice is shown to the user. This notice reminds the user to save his work. There are two buttons in the notice. The button **`Save now`** calls the function `Save`. The other button resets and enables the timer. Since the functions to manipulate the timer are not recursive and does not do any pattern matching, we use macro's.

```
resetTimer io :== timerOff (SetTimerInterval TId time io)
timerOn    io :== EnableTimer  TId io
timerOff   io :== DisableTimer TId io

remaindSave :: TimerState ProgState IO -> (ProgState,IO)
remaindSave _ s io
 # (button,io) = OpenNotice (Notice ["Save now?"] (NoticeButton lid "Later")
                                                   [NoticeButton sid "Save now"])
                 (DisableTimer TId io)
 | button==sid = Save s (resetTimer io)
               = (s,timerOn (resetTimer io))
where [lid,sid:_] = [0..]
```

Finally, there is an instance of `+` for strings used above and some constants. The first three constants determines properties of the drawing window. The value `time` determines the time interval between save remainders. Finally there are some id's for windows, timers and menu(item)s.

```
instance + {#Char} // + for String
where (+) s t = s +++ t

PictDomain      :== ((0,0), (1000,1000))
MinWindowSize   :== (50,50)
InitWindowSize  :== (500,300)
time            :== 5*60*TicksPerSecond     // Time between save remainders
DoNotCareId     :== 0
WId             :== 1
TId             :== 2
```

This completes our line drawing example. It demonstrates how all parts introduced above can be put together in order to create a complete program. It is tempting to add

features to the program in order to make it a better drawing tool. For instance to switch on and off the save remainder and set its time interval. An option to set the thickness would be nice, as well as circles rectangles etcetera etcetera. Adding these things requires no new techniques. In order to limit the size of the example we leave it to the user to make these enhancements. Chapter II.4 discusses a more sophisticated drawing tool.

5.7 Exercises

5.1 Write a program that applies a given transformation function from character lists to character lists on a given file. Structure the program such that the transformation function can be provided as an argument. Test the program with a function that transforms normal characters into capitals and with a function that collects lines, sorts them and concatenates them again to a character list.

5.2 Combine the `FileReadDialog` and `FileWriteDialog` functions into a complete copyfile program which copies files repeatedly as indicated in a dialog by the user.

5.3 Adapt the program you made for exercise 5.1 such that it transforms files as indicated in a dialog by the user.

5.4 Adapt the display file program such that the user can save the viewed file with a `SelectOutputFile` dialog. Use (possibly a variant of) the function `FileWriteDialog`. In order to assure that saving is done instantly instead of lazily the `Files` component of the `ProgState` can be made strict by prefixing `Files` in the type definition of `ProgState` with an exclamation mark. Add the text as a field in the state record. It may also prove to be useful to add the name of the file and the file itself to this state. In order to allow the user to overwrite the displayed file the program will have to be changed to used `fopen` for displaying instead of `sfopen` since a file opened with `sfopen` can be neither updated nor closed.

5.5 Adapt the program you made for exercise 5.3 such that it shows the result of a transformation of a file in a window such that the user can browse through it before saving it.

5.6 Include in the program of exercise 5.5 a menu function opening a dialog with `RadioItem`s such that the user can select the transformation to be applied.

5.7 Adapt the display file program such that the user can choose with a `ScrollingList` the font which is used to display the file.

5.8 Include in the program of exercise 5.6 a timer that scrolls to the next page automatically after a period of time which can be set by the user via an input dialog.

5.9 Extend an existing program using the function `GetCurrentTime` and a timer to display the time in hours and minutes every minute. Choose your own way to display the time: in words or as a nice picture using the draw functions from the I/O module `deltaPicture`.

5.10 (Large exercise) Extend the display file program with editing capabilities by extending the keyboard and mouse functions. Incorporate the results of exercises 5.5, 5.7 and 5.8 and extend it into your own window-based editor.

5.11 Change the line drawing program such that only horizontal and vertical lines can be drawn if the shift key is pressed during drawing. The line draw should be the 'best fit' of the line connecting the stating point and the current mouse position.

5.12 Extend the line drawing program such that the thickness of lines can be chosen from a sub-menu.