The Napier88 Standard Library Reference Manual

Version 2.2

Compatible With Napier88 Release 2.0

June 1994

Graham Kirby
Fred Brown†
Richard Connor
Quintin Cutts
Alan Dearle†
Vivienne Moore
Ron Morrison
Dave Munro

University of St Andrews, North Haugh, St Andrews, Fife KY16 9SS, Scotland.
†University of Adelaide, GPO Box 498, Adelaide, South Australia 5001, Australia.

This document should be referenced as:
"The Napier88 Standard Library Reference Manual Version 2.2".
University of St Andrews Technical Report CS/94/7.

Contents

1 Introduction

This manual describes the contents of the persistent store as supplied with Napier88 Release 2.0. These contents are known as Version 2.2 of the Napier88 Standard Library.

This manual does not describe the Napier88 language, which is described in the Napier88 Reference Manual (Release 2.0) [MBC+94a].

Version 2.2 of the Napier88 Standard Library is structured differently from the standard environment of Napier88 Release 1.0 and contains considerable additional software.

1.1 Accessing the Persistent Store

The persistent store may be accessed from Napier88 programs by calling the predefined procedure:

PS : **proc**(→ **any**)

The result returned is the *persistent root* injected into the union type *any*. Its type may vary between different persistent stores. In the Napier88 Release 2.0 store the persistent root is an environment initially containing the following environments:

name	**environment contents**
Error	error handling procedures which are called when errors occur during the execution of Napier88 programs
External	facilities provided by other sites
Library	standard procedures and other data which may be used in Napier88 programs
User	available for user data

Table 1.1: Environment contents

The initial structure of *Error* and *Library* is described in detail in this manual. *User* and *External* are local to a particular installation and users should consult the local administrator for details. The majority of data items in the standard library are constant and may be used but not updated by user programs. There are also some that may be updated in order to affect the behaviour of the system. The items in the library fall into a number of categories:

- procedures for compiling Napier88 programs;
- procedures for browsing the persistent store;
- procedures for performing I/O;
- procedures for constructing graphical user interfaces;
- procedures for controlling concurrent threads;
- procedures for accessing other Napier88 stores; and
- other utilities.

The initial environment structure of the standard library is shown in Figure 1.1:

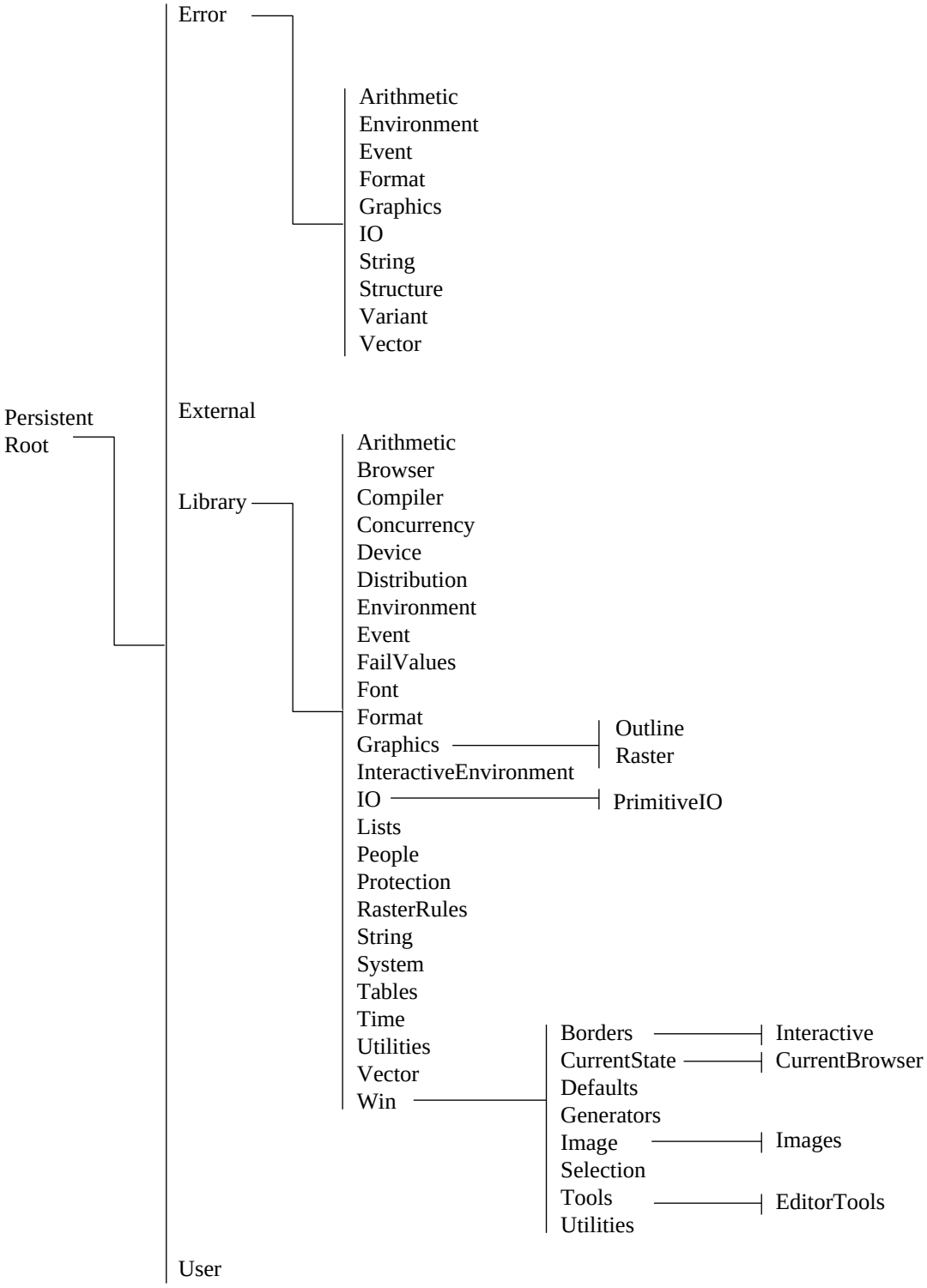


Figure 1.1: Initial environment structure in the standard library

2 The Napier88 Programming Environment

Part of the standard library is an integrated programming environment, written in Napier88, which allows the user to compose and execute Napier88 programs and examine their effects on the persistent store. Section 4.13 describes how to start up the integrated programming environment.

The programming environment supports *hyper-programming*, allowing the Napier88 source programs to contain embedded direct references to values, locations and types in the persistent store. The concept of hyper-programming is described in [FDK+92, Kir92, KCC+92b, KC93, MBC+94b].

The programming environment provides several varieties of window:

- multiple hyper-program windows;
- a compilation error display window;
- a browser window;
- multiple declaration set windows; and
- an output window.

The facilities provided by each window variety are now described.

2.1 Hyper-Program Windows

Hyper-program windows may be created by selecting *New Editor* from the background menu described in Section 2.5. Each window contains a hyper-program text editing area, a scroll bar and a row of light-buttons. The following operations are available in the text area:

operation	method
enter text	type at keyboard
position insertion point	click mouse button 1
set current selection	drag region of text with mouse button 1
extend current selection	click with mouse button 2
select word	double click with mouse button 1
delete current selection	type ‘backspace’ or ‘delete’
cut current selection	type ‘ctrl-x’
copy current selection	type ‘ctrl-c’
paste	type ‘ctrl-v’
insert hyper-program link	type ‘ctrl-l’
evaluate selected text	type ‘ctrl-e’

Table 2.1: Operations in hyper-program text area

An example of a hyper-program window is shown in Figure 2.1:

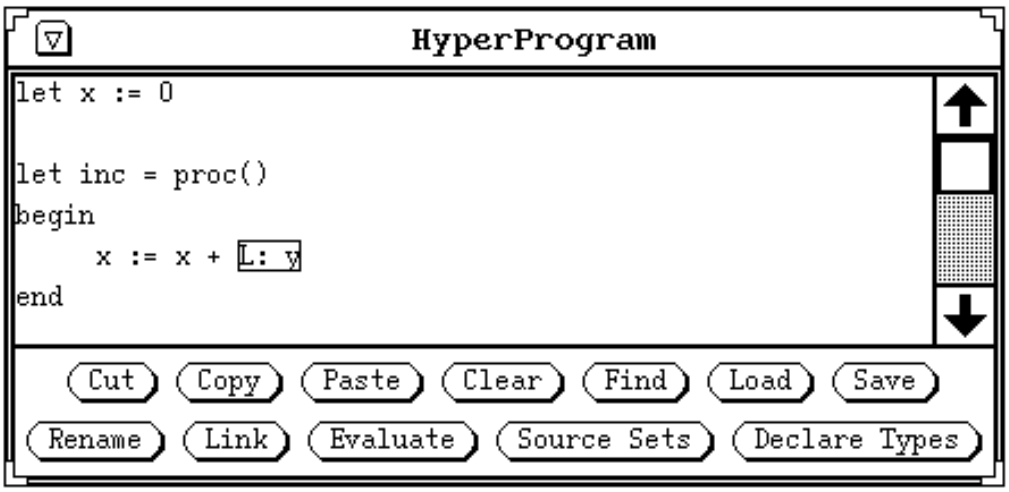


Figure 2.1: A hyper-program window

The operations available via the light-buttons are as follows:

operation	action	keyboard equivalent
Cut	This cuts the current selection into a buffer shared among all other hyper-program windows.	ctrl-x
Copy	This copies the current selection into the shared buffer.	ctrl-c
Paste	This replaces the current selection with the contents of the shared buffer.	ctrl-v
Clear	This deletes the entire contents of the editor.	
Find	This presents a dialogue allowing searching of the text for a given fragment of hyper-program text, either forwards or backwards from the end of the current selection.	
Load	This presents a dialogue allowing text to be loaded into the editor from a file.	
Save	This presents a dialogue allowing the textual contents of the editor to be saved to a file.	
Rename	This presents a dialogue allowing the currently selected light-button to be renamed. If no light-button is currently selected it has no effect and the dialogue is not displayed.	
Link	This inserts a hyper-program link to the currently selected value, location or type. A light-button representing it is inserted into the hyper-program text. The initial label on the button is a string beginning with the characters "V: ", "L: " or "T: " respectively. The rest of the label is the name, if any, associated with the selection (as, for example, when it is a location). The value, location or type associated with a button can be displayed in the browser window by pressing the button with mouse button 1.	ctrl-l
Evaluate	This attempts to compile the currently selected hyper-program text, executes the result if successful, and displays any result in the browser window. If the currently selected hyper-program text is a valid type expression then a representation of that type is displayed in the browser window. If a compilation error occurs the compilation error window is displayed (see Section 2.2).	ctrl-e
Source Sets	This displays a dialogue allowing the source declaration sets to be set (see Section 2.4.4).	
Declare Types	This attempts to compile the currently selected hyper-program text and adds any type declarations in scope at the end of the compilation to a selected declaration set (see Section 2.4.2). Existing declarations with the same names are over-written.	

Table 2.2: Light-button operations in hyper-program window

2.2 The Compilation Error Window

The *Compilation Error* window is displayed when compilation errors are encountered in a hyper-program. One sub-window shows the source code with the region of the first error highlighted. The second sub-window shows a message describing the error. When multiple errors are detected the *Next* and *Previous* buttons can be used to scroll through the errors. An example is shown in Figure 2.2:

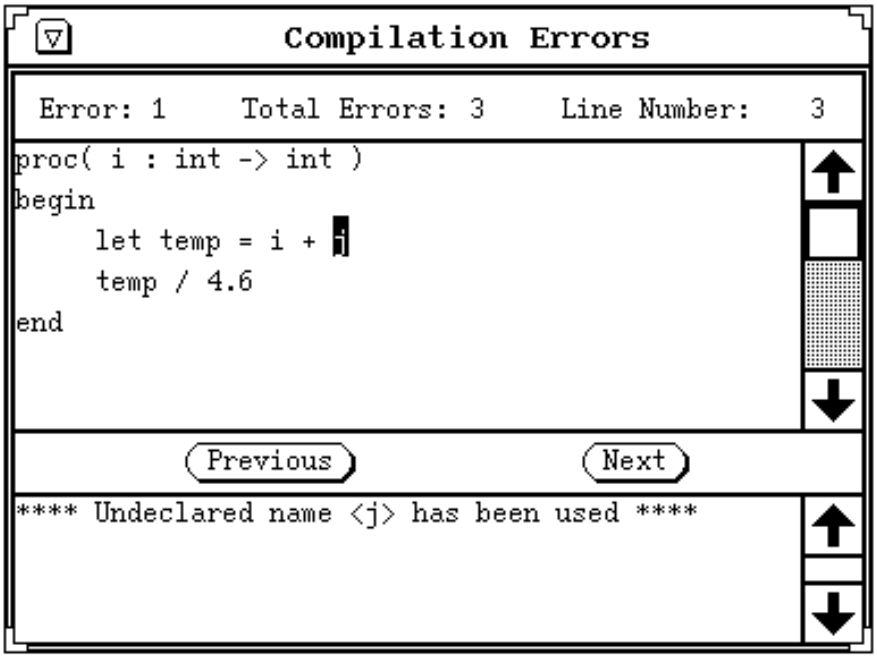


Figure 2.2: The compilation error window

2.3 The Browser Window

The *Browser* window is displayed automatically when the programming environment first starts up. It is used to display representations of values produced by the evaluation of hyper-programs. The root structure of the persistent store can be displayed by selecting *Show PS* from the background menu described in Section 2.3.13.

The form in which a value is represented depends on the type of the value. Integers, reals, strings and booleans are written to the output window. The manner in which other types are displayed is described in Sections 2.3.2–2.3.10.

2.3.1 Operations on Windows

The following operations are available on all windows displayed in the browser window:

- The window can be selected or deselected by clicking on the border with mouse button 1. If the window is not already selected it becomes selected and any other selected windows are deselected. If the window is already selected it becomes deselected. When a window is selected the corresponding value is also considered to be selected. This is of relevance when inserting links into hyper-programs and when selecting certain operations from the browser background menu which operate on the selected value.

- The window can also be selected or deselected by clicking on the border with mouse button 2. In this case other windows are unaffected.

- A menu can be obtained by holding down mouse button 3 on the border. The entries in the menu are *Front*, to bring the window to the front, *Back*, to put the window to the back, and *Dismiss*, to undisplay the window.

2.3.2 Environments

To show an environment the browser displays a menu window containing an entry for each binding in the environment. For base type values the corresponding entry shows the type while for instances of constructed types only the type constructor is shown. An example is shown in Figure 2.3:

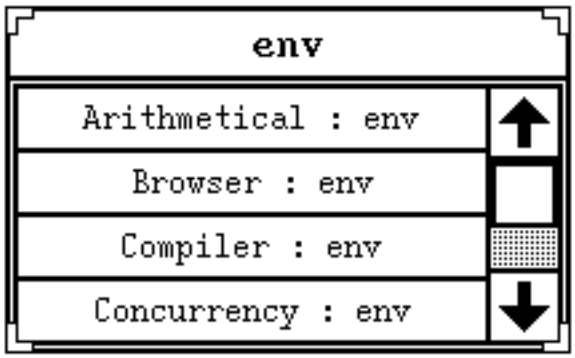


Figure 2.3: An environment menu

The operations available on an environment menu entry depend on the mouse button used:

- Clicking with mouse button 1 results in the menu entry being highlighted and any other highlighted menu entries or windows being un-highlighted. The corresponding environment location is selected.

- Clicking with mouse button 2 results in the menu entry being highlighted while other selected menu entries or windows are unaffected.

- Holding down mouse button 3 results in a pop-up menu being displayed. Selecting *Show* results in the value of the corresponding environment binding being displayed in the browser. If the value is of such a type that a new window is displayed for it, an arrow is drawn from the menu entry to the new window as shown in Figure 2.4. Selecting *New Universe* also results in the value being displayed but in a separate universe as described in Section 2.3.11.

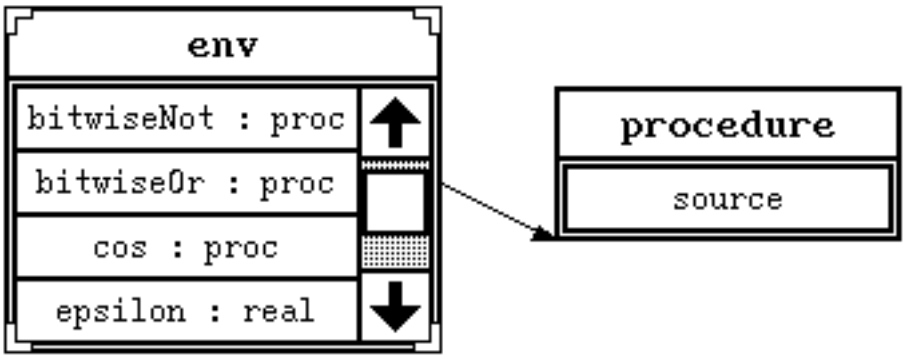


Figure 2.4: Link from environment location to value

2.3.3 Structures

Structures are displayed in the same way as environments. An example of a structure menu is shown in Figure 2.5:

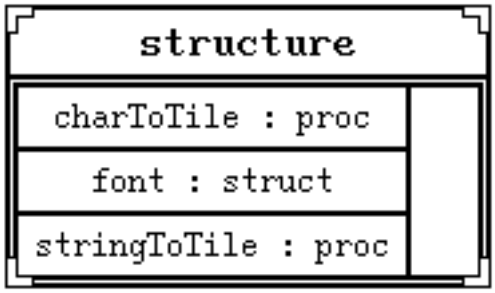


Figure 2.5: A structure menu

2.3.4 Variants

Variants are displayed in the same way as structures except that the entry for the branch of the variant which is actually present is indicated by the prefix '+'. Selecting other entries has no effect. An example of a variant menu is shown in Figure 2.6:

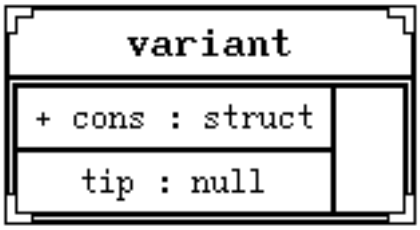


Figure 2.6: A variant menu

2.3.5 Vectors

Vectors are displayed in the same way as structures with an entry for each element of the vector: Each entry shows the corresponding index number. An example of a vector menu is shown in Figure 2.7:

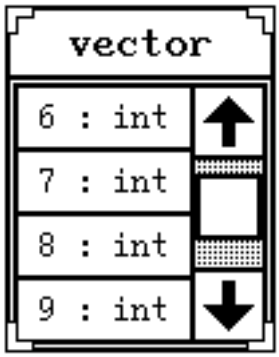


Figure 2.7: A vector menu

2.3.6 Images

An example of an image window is shown in Figure 2.8:

Figure 2.8: An image window

The size of the image in pixels is shown in the bottom right corner of the window.

2.3.7 Pictures

Pictures are displayed in a similar way to images, as shown in Figure 2.9:

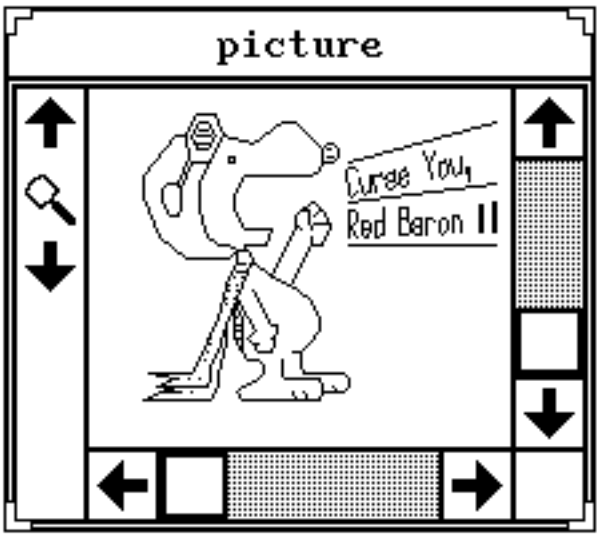


Figure 2.9: A picture window

When a picture is first drawn it is scaled so that it fits completely into the display area. Resizing the window enlarges or reduces the display area but does not alter the scale of the picture. The scroll bars can be used to pan the display area to a different region of the picture. The window also contains two arrow buttons on the left hand side which can be used

to enlarge or reduce the scale at which the picture is drawn. This makes it possible to zoom in on a region of interest or to move back to view the picture as a whole.

2.3.8 Procedures

To show a procedure the browser displays a menu with a single entry *source*. When this entry is selected using any mouse button the browser displays a hyper-program window containing the source code for the procedure. The source code may be selected and copied but not altered. If the procedure does not have source code attached the browser displays a message to this effect in the output window. An example of a procedure menu is shown in Figure 2.10:

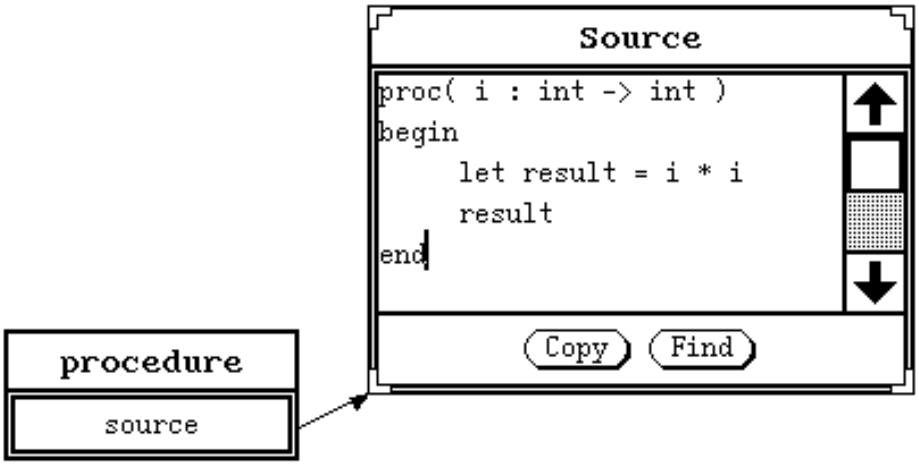


Figure 2.10: A procedure window

2.3.9 Abstract Datatypes

To shown an instance of an abstract datatype the browser displays a menu with no entries. An example of an abstract datatype menu is shown in Figure 2.11:

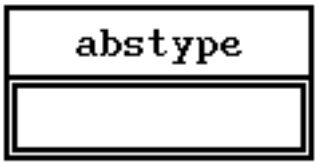


Figure 2.11: An abstract datatype window

2.3.10 Files

To show a file the browser displays its name in the output window.

2.3.11 Universes

The screen may become cluttered when the user browses a large data structure. **Universes** can be used to organise the data space. A universe is created by selecting *Universe* from the menu brought up with mouse button 3 over a structure, variant, vector or environment menu. A new browser sub-window is then created and the corresponding value displayed inside it. An example of a universe window is shown in Figure 2.12:

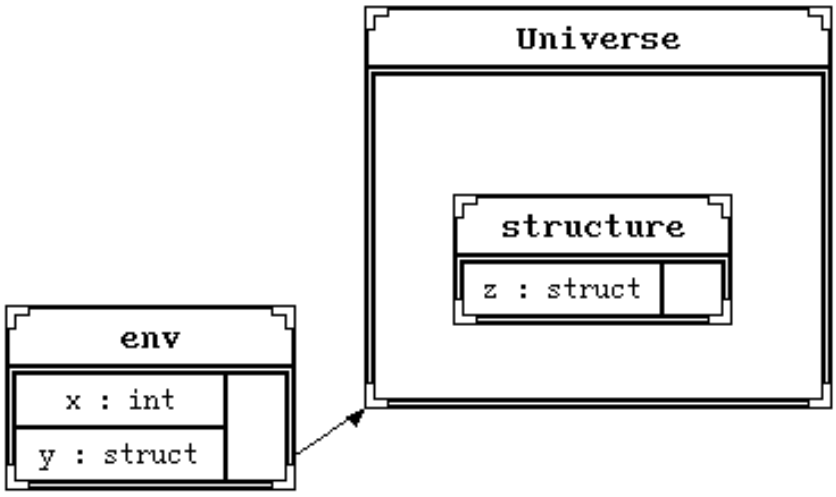


Figure 2.12: A universe window

The new value, the structure with field *z* in the example, is displayed by a new invocation of the browser which operates entirely within the universe window. Any other objects discovered from that new object will be confined to the window. In this way the object and others accessible from it are kept separate from the rest of the visible data. Universes also provide a grouping mechanism in that all the objects in a universe can be moved or deleted in one action by operating on the window containing them. Any number of universes can be created, and they can be nested to any degree.

2.3.12 Types

A representation of the type of a value in the browser window may be obtained by selecting the corresponding window and selecting *Show Type* from the browser background menu. The browser displays a window containing a canonical string representation of the value's type. Note that a type representation is displayed only when a window is selected, not when a menu entry is selected. This is in order to avoid confusion between the contents of a location and its type. An example of a type representation is shown in Figure 2.13:

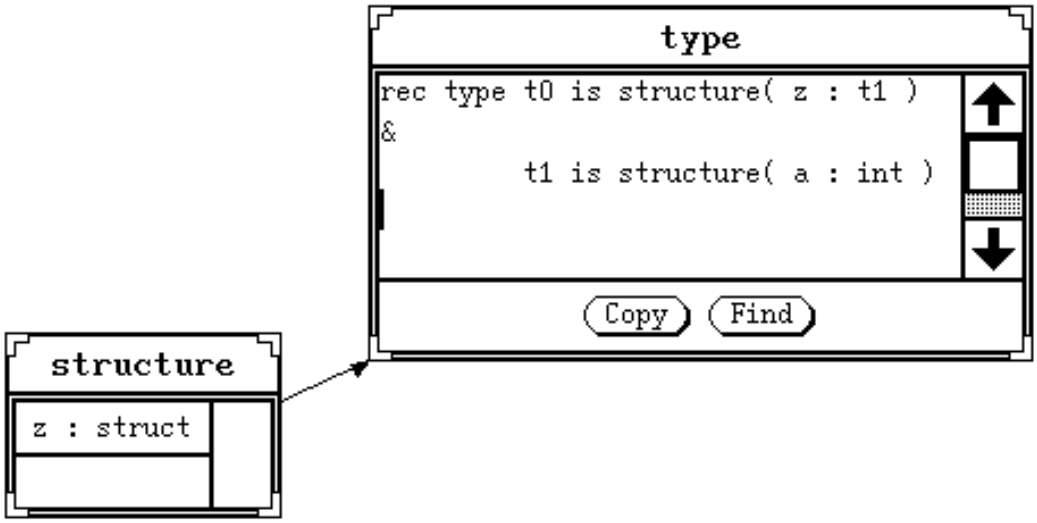


Figure 2.13: A type representation

The browser also displays a representation of a type linked into a hyper-program when the corresponding light-button in the hyper-program window is pressed. In this case the representation may be a canonical string as above or, where type constructor information is

available, the original source code is displayed as a hyper-program fragment. An example of a type constructor source representation, with a hyper-program link to a component type, is shown in Figure 2.14:

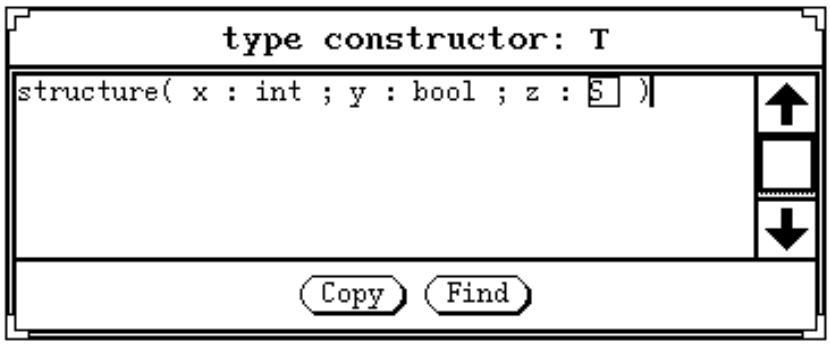


Figure 2.14: A type constructor representation

2.3.13 The Browser Background Menu

A background menu may be obtained in the browser window by holding down mouse button 3. The menu provides the following operations:

operation	**action**
Centre	pans the browser window so that the first selected object is in the centre of the window
Show PS	displays the root of persistence
Show Type	displays a canonical representation of the type of the selected value
Clear	removes all objects displayed in the browser window

Table 2.3: Browser background menu operations

2.3.14 The Panning Tool

The *Panning Tool* window allows the browser window to be panned over the unbounded view space. The circle represents a joy-stick which can be dragged using mouse button 1. While the joy-stick is off-centre the browser window pans in the same direction. The panning increments are proportional to the amount the joy-stick is displaced from the centre. The Panning Tool window is shown in Figure 2.15:

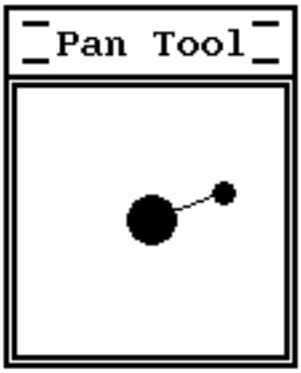


Figure 2.15: The panning tool

2.4 Declaration Sets

The user may create *declaration sets* containing named values, locations and types to use in future program evaluation. Each declaration set has a unique name and may be thought of as forming an additional outer scope for a program. Free identifiers in a program are resolved by scanning the declaration sets associated with the program.

A type entry in a declaration set may represent either a type only, or a type constructor. Which is obtained depends on the method used to create the entry. Both type and type constructor names may be used as type denotations in programs, but only type constructor names may be used to construct instances of types.

The declaration sets model is based on a number of earlier systems: Napier88 Release 1.0 [MBC+89a]; ABERDEEN [Far91]; and a previous version of the Napier88 programming environment [KCC+92a].

The operations on declaration sets are:

- create a new declaration set;
- delete a declaration set;
- add a value, location or type to a declaration set;
- display the contents of a declaration set; and
- choose an ordered list of declaration sets to use for compilation.

2.4.1 Creating and Deleting Declaration Sets

Creation and deletion of declaration sets is performed using the declaration sets menu obtained by selecting *Declaration Sets* from the background menu described in Section 2.6. The declaration sets menu is shown in Figure 2.16:

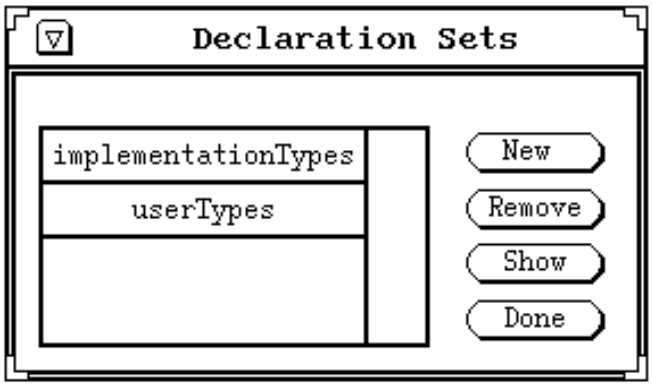


Figure 2.16: The declaration sets menu

The menu contains a list of the existing declaration sets and light-buttons providing the following operations:

operation	**action**
New	displays a dialogue prompting for a name for the new declaration set. If the new name clashes with an existing declaration set name an error message is displayed and the dialogue is re-presented. Otherwise a new empty declaration set is created and the list is updated.
Remove	permanently removes any selected declaration sets
Show	displays the contents of any selected declaration sets
Done	undisplays the declaration sets menu

Table 2.4: Declaration sets menu operations

2.4.2 Adding To a Declaration Set

One method of adding a value, location or type to a declaration set involves the user selecting the corresponding representation in the browser window. The user then selects *Add to Declaration Set* from the background menu described in Section 2.6. If a named location is selected that name is used; otherwise the user is prompted for a name. A dialogue then allows the user to choose an existing declaration set or to create a new one. If an entry with the same name already exists in the chosen declaration set that entry is overwritten by the new one.

The user may add a group of type declarations to a declaration set by pressing the *Declare Types* button in a hyper-program editor. This attempts to compile the selected code, or the entire contents if none is selected, and if successful adds all the top-level type definitions to a declaration set chosen as above.

2.4.3 Displaying a Declaration Set

The contents of a particular declaration set may be displayed by pressing the *Show* button in the declaration sets menu described in Section 2.4.1. This displays a further menu for each selected declaration set. An example is shown in Figure 2.17:

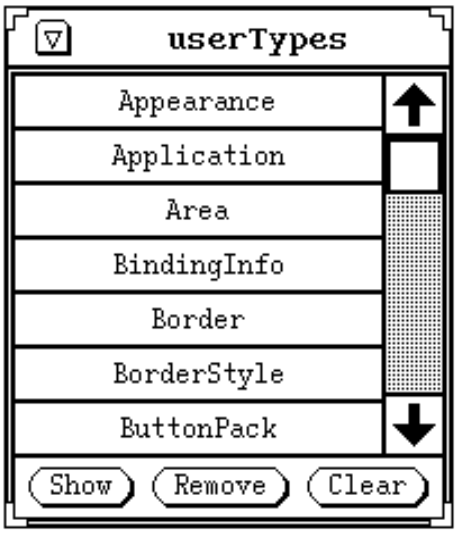


Figure 2.17: A declaration set menu

Each menu contains a list of the entries in that declaration set and light-buttons providing the operations shown in Table 2.5:

operation	**action**
Show	displays any selected entries in the browser window
Remove	permanently removes any selected entries from the declaration set
Clear	permanently removes all entries from the declaration set

Table 2.5: Declaration set menu operations

A hyper-program link to an entry in a declaration set can be inserted into a hyper-program by selecting the appropriate menu entry and pressing the *Link* light-button as described in Section 2.1.

2.4.4 Choosing Declaration Sets

The user may associate a particular combination of declaration sets with a hyper-program editor. These declaration sets are then used in evaluating program fragments in that editor. When it is first created an editor has no declaration sets associated with it. Declaration sets may be added by pressing the *Source Sets* light-button. This displays a dialogue as shown in Figure 2.18:

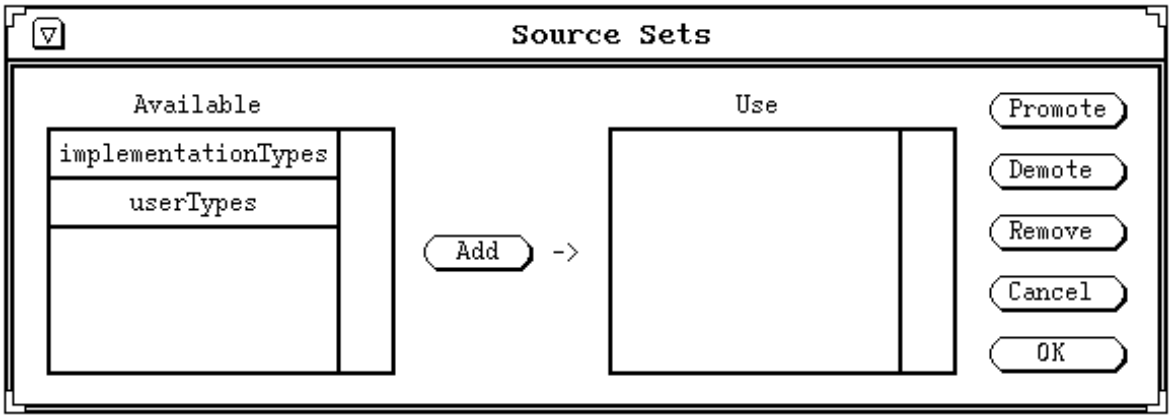


Figure 2.18: Dialogue for setting source declaration sets

The *Available* list on the left shows all the existing declaration sets. The *Use* list on the right shows those currently associated with the editor, scope level increasing down the list. If two declaration sets associated with an editor both contain an entry with the same name, the one in the declaration set nearer the top of the list will mask the other. This is analogous to normal scoping rules.

The dialogue also contains light-buttons providing the following operations:

operation	**action**
Add	adds any selected declaration sets in the *Available* list to the bottom of the *Use* list
Promote	moves the selected declaration sets up one position in the list if possible
Demote	moves the selected declaration sets down one position in the list if possible
Remove	removes any selected declaration sets from the use list
Cancel	undisplays the dialogue without altering the declaration sets associated with the editor
OK	undisplays the dialogue and associates the chosen declaration sets with the editor

Table 2.6: Source declaration sets menu operations

2.5 The Output Window

The *Output* window displays messages from the browser. Its contents may be scrolled and cleared but not edited. A procedure to write messages to the Output window is initially available in the *CurrentState* environment described in Section 4.26.2.

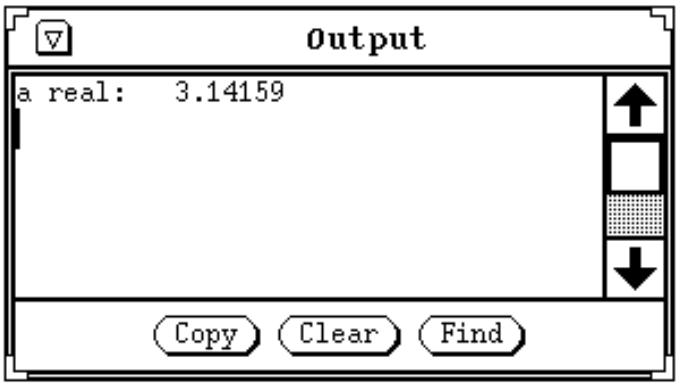


Figure 2.19: The output window

2.6 The Background Menu

A background menu may be brought up in the programming environment by holding down mouse button 3. The menu provides the following operations:

operation	**action**
New Editor	creates a new hyper-program editor window and displays it
Add to Declaration Set	for each currently selected browser value, location or type, adds a binding to a declaration set as described in Section 2.4.2
Declaration Sets	displays the declaration set menu as described in Section 2.4.1
Show All Windows	displays all windows registered with the programming environment as described in Section 2.7
Delete Non-Visible	unregisters windows not displayed, as described in Section 2.7
Stabilise	calls the *stabilise* procedure described in Section 4.20
Quit	quits the programming environment

Table 2.7: Background menu operations

2.7 Persistent Windows

Programming environment windows persist between sessions of the programming environment. When the environment is shut down the positions and sizes of the windows are recorded and restored when it is next started up.

When a window is undisplayed by selecting *Dismiss* from its border menu, it is still retained by the programming environment. The user can cause all such windows to be redisplayed by selecting the entry *Show All Windows* from the background menu. It is possible however to remove editor windows permanently from the programming environment by selecting the entry *Delete Non-Visible*. This has the effect of deleting any editor windows not currently displayed.

2.8 Accessing the Current State

To facilitate access to the programming environment from programs the following values are available while the environment is active:

currentBuffer : Editor[Binding]
currentError : **proc**(**string**)
currentOutputPack : EditorPack[Binding]
currentWindowManager : WindowManager
currentWriteString : **proc**(**string**)
CurrentBrowser : **env**

These are described in Section 4.26.2.

2.9 Using Napier88 from UNIX

There are a number of commands that control the execution of the Napier88 system from UNIX.

2.9.1 nps: Compiling Type Declarations

For convenience, when a program is compiled it may be compiled against a set of pre-compiled type declarations. This command is used to save such a set of type declarations and is compatible with the declaration sets described in Section 2.4.

The source file must consist purely of type declarations or the command will fail and an error message will be displayed. The general form of the command is:

*nps [sourceFile declarationSet] [-l] [-t declarationSet]**

The first two parameters are the name of a file containing the type declarations and the name of a declaration set. The options are:

-l (list) Produce a source listing.

-t (types) Compile against existing sets of type declarations. This option may be repeated. The first declaration set forms the outermost scope and the source file forms the innermost scope.

For example, to save a set of type declarations given in the file *types1.N* in the declaration set *types1*, the following could be used:

nps types1.N types1

To save a further set of type declarations compiled against this set, with a listing, the following could be used:

nps types2.N types2 -l -t types1

If a source file is not specified as a parameter, the command enters interactive mode. First the command prompts for a list of declaration sets against which a source file may be compiled. Each declaration set is at an inner scope level with respect to any previously specified sets. To finish specifying source declaration sets, *return* is entered in response to the command's prompt *Source type set:*

Once any existing declaration sets have been specified, the command prompts for a source file to be compiled and the declaration set name under which the type declarations should be saved. The source file is compiled against all the declaration sets that have been specified. When the type declarations have been saved the command prompts for another source file to compile. To finish saving new declaration sets, *return* is entered in response to the command's prompt *Filename:*

To interactively save a set of type declarations, given in the file *types.N*, in the set *sometypes*, the following could be used:

nps	! the command
Source type set: <*return*>	! request for a declaration set to compile against;
	! none to specify
Filename: *types.N*	! the source file to compile
Destination type set: *sometypes*	! request for new declaration set name
Filename: <*return*>	! no more source files to be compiled

For backward compatibility with PS-algol implementations of Napier [MBC+89b], the non-interactive version of *nps* allows a database name and password to be specified with each declaration set name; these are ignored.

2.9.2 npc: Compiling Programs

This command is used to compile Napier88 programs and is parameterised by the name of the source file. The source file must contain a void sequence [MBC+94a] or the command will fail and an error message will be displayed. The general form of the command is:

*npc [sourceFile] [-els] [-t declarationSet]**

The options are:

-e (execute) Execute the program if the compilation succeeds.

-l (list) Produce a source listing.

-s (silent) Do not produce an object code file.

-t (types) Compile against existing sets of type declarations. This option may be repeated. The first declaration set forms the outermost scope and the source file forms the innermost scope.

For example, to compile a program in a file *prog.N* without executing it, without a listing, producing an object code file and using two existing type declaration sets, the following could be used:

npc prog.N -t outerTypes -t innerTypes

In this case the result of the compilation is placed in a file *prog.out*. In cases where the source filename does not end in *.N* the filename is constructed by appending *.out* to the source filename.

To compile a program in a file *prog.N* and execute the result without producing an object code file the following could be used:

npc prog.N -es

If a source file is not specified as a parameter, the command enters interactive mode. First the command prompts for a list of declaration sets against which a source file may be compiled. Each declaration set is at an inner scope level with respect to any previously specified sets. To finish specifying declarations sets, *return* is entered in response to the command's prompt *Source type set:*

Once any existing declaration sets have been specified, the command prompts for a source file to be compiled. The source file is compiled against all the declaration sets that have been specified. When the compilation is complete the command prompts for another source file to compile. To finish compiling, *return* is entered in response to the command's prompt *Filename:*

To interactively compile the program in the file *prog.N*, against the type declarations in the set *sometypes*, the following could be used:

npc	! the command
Source type set: *sometypes*	! request for a declaration set to compile against
Source type set: <*return*>	! request for another declaration set; no more to specify
Filename: *prog.N*	! the source file to be compiled
Filename: <*return*>	! no more source files to be compiled

A database name and password may be specified, and ignored, as for *nps*.

2.9.3 npr: Running Programs

This command is used to run the Napier88 system. The general form is:

npr [objectFile]

The command may be parameterised by the name of a file containing a compiled Napier88 program to be executed. Otherwise the Napier88 system is restarted from the state preserved by the most recent stabilise operation.

For example, to execute the compiled version of the program *prog.N*, the following could be used:

npr prog.out

2.9.4 napier88: Starting the Programming Environment

This command is used to start up the interactive programming environment. The general form of the command is:

napier88 [-d screenDepth]

The optional parameter specifies the number of planes required in the display, subject to the limitations of the display device. The command calls the procedure *startProgrammingEnv* described in Section 4.13.

2.9.5 nprgc: Stable Store Garbage Collection

This command is used to perform a garbage collection of the stable store. No other programs may be run against the store while the garbage collection is taking place. For example,

nprgc

Note that executing this command may actually increase the size of the UNIX file which contains the stable store. See *nprcompact*.

2.9.6 nprstats: Stable Store Statistics

This command is used to display statistics about the stable store. No other programs may be run against the store while the statistics gathering is taking place. For example,

nprstats

```
maximum space :         383.609 Mbytes
allocated space :       5.581 Mbytes ( 85.7%)
unallocated space :     0.281 Mbytes (  4.3%)
unused space :          0.000 Mbytes (  0.0%)
management space :      0.654 Mbytes ( 10.0%)
number of objects :     85671 objects

The following configuration details are specified:
KEY_TO_ADDR
KEYS_ARE_INDIRECT
REQUEST_STABILISE
KEY_RANGE
minimum key :           64000
maximum key :           17fffff8
key alignment :         8
```

The first set of statistics reflect the current state of the stable store. The second set reports details of the particular stable store implementation in use.

2.9.7 nprcompact: Stable Store Compaction

This command is used to compact the UNIX file containing the stable store. No other programs may be run against the store while the compaction is taking place. For example,

nprcompact

2.9.8 nprformat: Stable Store Initialisation

This command is used to create a new empty stable store. If the stable store directory as described in Section 2.9.9 already contains a store, an error message is displayed and no further action is performed. For example,

nprformat

2.9.9 nprsethost: Setting the Host

This command is used to set the host from which programs may run against the stable store. The general form of the command is:

nprsethost [-n] [hostname]

By default the command may only be run from the host for which the store is currently set. If the *-n* flag is specified this constraint is over-ridden; this should be used with caution and only when there is no possibility that there is currently a program running against the store. If the hostname parameter is omitted the name of the current host is used.

2.9.10 nprregisterhost: Registering a New Host

This command is used to register hosts with the Napier88 system. It takes as a command line argument either the authorisation code for a single machine, or the name of a file containing multiple authorisation codes. For example,

nprregisterhost fj8ahd3h7a2
nprregisterhost auth.codes

2.9.11 UNIX Environment Variables

There are several shell variables that allow the Napier system to be dynamically configured. They are:

NPRDIR : this variable defines the pathname for the release directory. All the commands held in the *bin* directory use this variable to construct the pathnames of the executable programs to be run. By default this is */usr/lib/napier88*.

NPRSTORE : this variable defines the pathname for the UNIX directory containing the stable store file. By default this is the name of the processor prefixed by *$NPRDIR/sstore.* e.g. on a processor named *panda* the pathname would be *$NPRDIR/sstore.panda*. If the desired size of stable store is too large for the disk partition containing the release directory, a symbolic link can be used to map the store's pathname onto a larger disk partition.

NPRHEAP : this variable defines the size of the local heap (in megabytes) used by the Napier88 interpreter. By default this is 8.

3 Graphical User Interface Programming

This section gives an outline of the user interface programming facilities provided by the WIN window management system—which is also used to implement the programming environment described in Section 2.

3.1 Windows and Window Managers

The principal entities in WIN are *windows* and *window managers*. A window has two main functions: to display a bitmapped image and to handle user input. A window manager is used to display and manipulate windows. As well as providing program output facilities, windows may be used to implement user interface interactors such as light-buttons, sliders, menus etc.

Each window has encapsulated in it:

- an application procedure which processes input events received by the window; and
- an image on which raster operations may be performed by the application procedure or by other programs.

Windows exist independently of window managers. Since they are Napier88 values they have full civil rights and thus may be held in the persistent store, assigned to variables, passed as procedure parameters etc. All window operations may be accessed by programs independently of whether a window is displayed by a window manager. When a window *is* displayed by a window manager its image may become visible to the user. Its application procedure may also receive input events directed to the window, via the window manager, from the user.

Each window manager operates within a parent window. This recursive structure allows nesting of window managers to any depth. The recursion is grounded by a distinguished root window manager which operates directly on the display device.

The procedures for creating windows and window managers are described in Section 4.26.4. A window is displayed by calling one of a window manager's interface procedures, passing it the window and information describing where to position it. The details are also described in Section 4.26.4.

3.2 Window Attributes

A window has a number of attributes which may be read and set. These include:

- its size;
- its title;
- its minimum and maximum size;
- its behaviour when its size is changed—used to allow the window display to be redrawn appropriately;
- its application procedure—determining how the window handles input events;
- its border style—used by the window manager to show the outline of the window and to allow interactive window manipulation;
- its priority for receiving input events;

- the number of planes in its display image—affecting how many colours can be displayed; and
- the shape of the cursor when over the window.

The interface procedures which allow these attributes to be read and set are described in Section 4.26.4.

3.3 Drawing on Windows

The display of a window may be updated via raster operations on its bitmap. Raster operations may be performed between the window and another window or an image, in either direction. The four possibilities are:

source	*destination*
window	image
image	window
window	another window
another window	window

Table 3.1: Window raster operations

Straight line drawing on windows is also supported. The window drawing functions are described in Section 4.26.4.

3.4 Current and Selected Windows

Windows displayed in the programming environment may be distinguished in two ways: by being *current* and by being *selected*.

3.4.1 Current Windows

A window may be *current* with respect to the window manager that is displaying it. No more than one of the windows displayed by a particular window manager may be current. If there is a current window then any keyboard input events received by the window manager are directed to the application procedure of that window. If there is no current window then keyboard events are discarded.

A window may be made current by calling the *makeCurrent* procedure of the window manager displaying it. This is described in Section 4.26.4. A window may also be made current interactively, either by moving the cursor over it or clicking with a mouse button within it.

A current window may be distinguished by its border. The border styles *fixedX*, *menuX* and *variableX*, for example, indicate a current window by showing two parallel lines along the title bar. These are described in Section 4.26.1.

3.4.2 Selected Windows

Any number of windows may be *selected* with respect to the programming environment as a whole. The list of selected windows may be read by application programs and acted on accordingly. For example, a command available in a drawing application might change the size of all the selected windows.

A selected window may be distinguished by its border. The border styles *fixedX*, *menuX* and *variableX*, for example, indicate a selected window by showing an inverted area along the title bar. These are described in Section 4.26.1.

A window may be both current and selected simultaneously.

3.5 Applications and Input Events

Every window has an application, a procedure which processes input events received by the window. Those input events may be keyboard events, if the window is current, or mouse events within the window area. Input events are represented by instances of type *Event*:

```
type Event is variant( chars :            string;
                       mouse :            Mouse;
                       select, deselect : null )

type Mouse is structure( x,y : int ; buttons : *bool )
```

Keyboard events are represented by the *chars* branch. The string contains the characters typed since the last keyboard event was issued. The length of the string is ≥ 1 since keyboard events are generated only when keyboard input occurs. The time that must elapse between key presses in order for separate events to be generated is not defined. Keyboard events are generated only when keys are pressed down. No events are generated when keys are released.

Mouse events are represented by the *mouse* branch. The structure contains the coordinates of the mouse as two integers, and the state of the mouse buttons as a vector of booleans, each element of which is **true** iff the corresponding mouse button is currently depressed. Mouse events are generated repeatedly whenever there is no keyboard input. Consecutive mouse events may thus contain the same information. To reduce the rate of structure creation, a single mouse structure is used for all events. Where an application needs to retain the information in a mouse event it is necessary to copy the contents of the structure, rather than simply retaining a reference to the structure, since the contents will be overwritten when the next mouse event is generated.

Each time WIN sends an event to a window application it compares that application with the application that received the previous event. If they are different WIN sends a *deselect* event to the previous application and then a *select* event to the new one, before sending the current input event to the new application. Select and deselect events do not themselves carry user input but they enable an application to perform particular actions when it first becomes the focus of input and when it ceases to be the focus.

The type of an application is:

```
proc( Event )
```

By convention WIN applications do not perform busy waiting for input or call blocking IO procedures. If this convention is not observed applications in other windows may be prevented from receiving input directed to them.

3.5.1 Background Windows and Applications

By convention WIN applications do not call the raster operations of any window in which a window manager is running. If this convention is not observed the display areas of windows displayed by that window manager may be corrupted. It may be required, however, to draw on the background of a window manager, for example in an application that shows links between windows by drawing lines between them.

The facility is provided safely by allowing a window manager to display a window in the background. Only one window may be so displayed at a time and a background window is always behind all other windows, no matter what levels they are placed at. If the programmer wishes to be able to draw anywhere on the background of the window manager this can be done by creating a window the same size as the window containing the window manager, displaying it in the background and then drawing on that window.

Alternatively the programmer may wish to have an application running in the background of the window manager without the need to draw on the background. If a background window is used there is an unnecessary memory overhead involved in storing the contents of the window. It is possible to set a background application which processes any events not dealt with by window applications.

3.6 Interface Interactors

The WIN library provides a number of pre-defined user interface interactors from which a user interface may be composed. Each interactor is a window; interfaces are built up by displaying interactor windows together in a parent window. The types of interactors provided are:

- light-buttons
- sliders
- menus
- check boxes
- radio buttons
- hyper-text editors

Various varieties of each interactor type may be created; the generator procedures are described in Section 4.26.7.

3.7 The User Interface Editor

The user interface editor allows the programmer to create WIN user interfaces interactively rather than textually. This aids both initial coding and later adjustment of an interface. It was inspired by Luca Cardelli's paper *Building User Interfaces by Direct Manipulation* [Car88].

3.7.1 Interface Design

For the purposes of the editor, an interface is a collection of interactors generated using the WIN library. Interactors are items such as menus, check-boxes, light-buttons etc.

The design of an interface involves the selection of the appropriate interactors to give the desired functionality and layout of items . Having specified this, an instance of the interface can be generated by the user specifying what happens when, for example, a light-button is pressed or a menu option selected. The collection of interactors comprising an interface is implemented as a single *root window* which can then be displayed by any window manager.

For example a simple painting tool could be implemented by the root window shown in Figure 3.1:

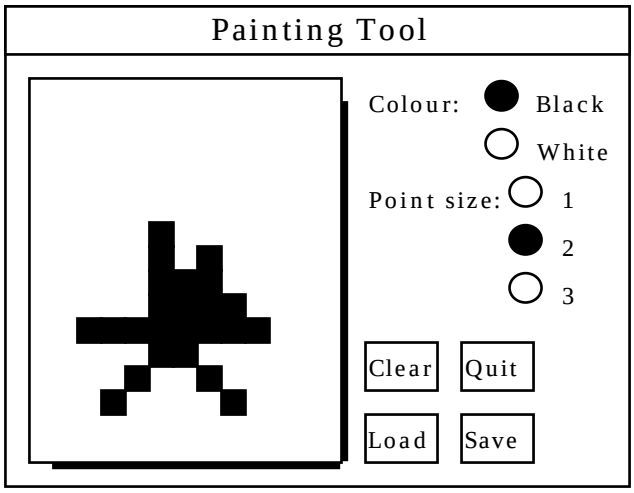


Figure 3.1: A root window

A window interactor is used for the drawing area, and light-buttons and groups of radio buttons provide the means of control.

3.7.2 Nested Windows

Windows within interface designs may be nested using *manager* interactors. Each one consists of a window and associated window manager allowing sub-windows to be contained.

As an example of the use of nested windows, consider the painting tool example introduced above. Imagine that the light-buttons and radio button groups are contained within a control panel window (displayed with an invisible border). The hierarchy of this interface would be as shown in Figure 3.2:

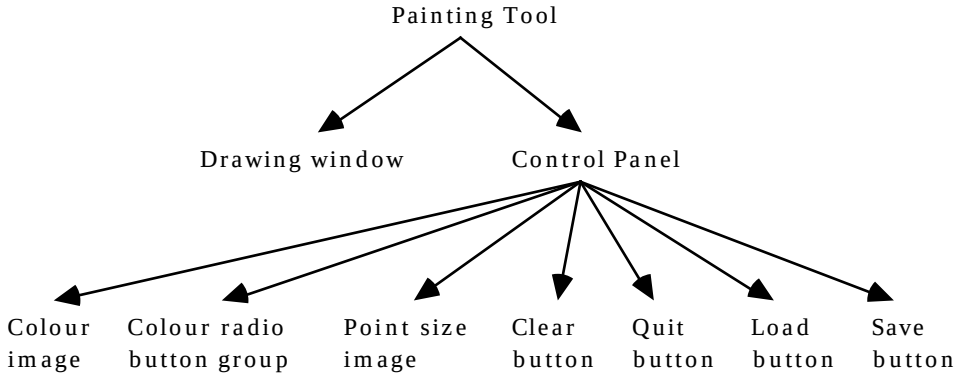


Figure 3.2: An interface hierarchy

Here the Control Panel could be implemented with a *manager* interactor.

3.7.3 Running the Editor

A window containing an interface editor may be created by calling the generator procedure described in Section 4.14. The resulting window has a fixed size of 1000 by 700 pixels.

The editor window consists of two sections, a control panel and the main editor window in which designs currently being edited are displayed. This layout and the contents of the control panel are illustrated in Figure 3.3:

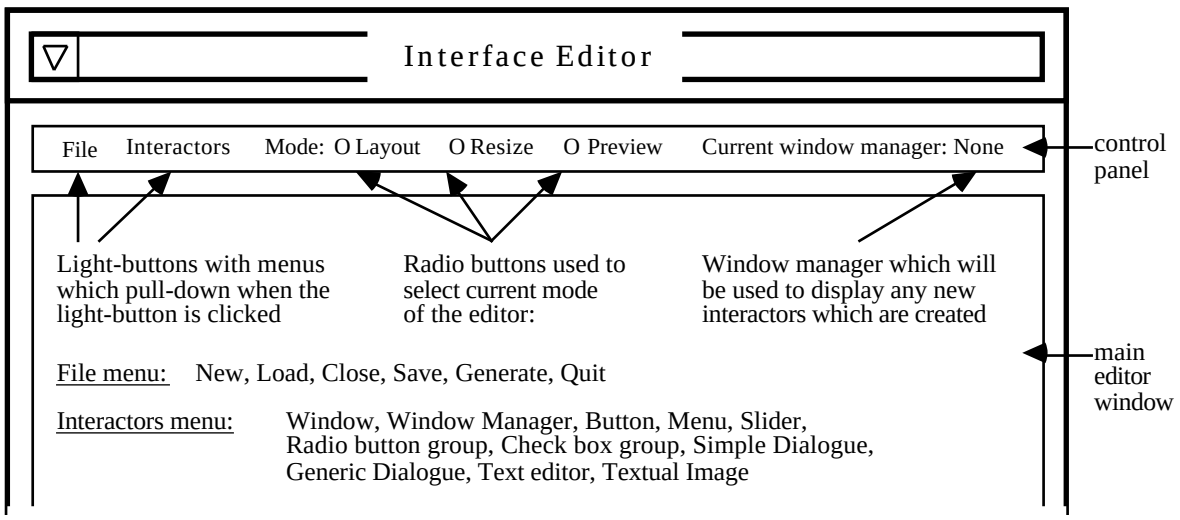


Figure 3.3: An interface editor window

3.7.4 Mode Buttons

Clicking on one of these changes the current mode. At present only *Layout* and *Preview* are implemented but *Resize* is included in anticipation of future implementation.

The default mode is *Layout* in which designs may be constructed and laid out.

Preview mode allows the user to see what the interface will look like in its final state and experiment with it, i.e. click on light buttons, select menu entries etc.

3.7.5 The Current Window Manager

The *current window manager* is the window manager into which any new interactor is placed. It can be associated with either the root window of a design or a manager interactor within a design. Initially the current window manager is unspecified.

The current window manager setting is displayed in the control panel. This is a textual representation, akin to a pathname. For example "Test/ControlPanel" would refer to the manager interactor titled "Control Panel" inside design "Test".

The current window manager setting can be changed in one of four ways:

- when a new design is created or an existing one loaded;
- when a manager interactor is created;
- when the current window manager is 1) associated with the root window of a design which is closed or 2) associated with a manager interactor which is deleted—in either case, the program tries to select the most sensible current window manager; or
- by clicking with the right mouse button inside a root window or manager interactor inside a design—the current window manager will be set to the associated window manager and the field on the control panel will change to reflect this.

3.7.6 File Menu

Options in this menu allow the user to start working on new designs, close current ones, load and save designs and quit from the program (*Generate*, the remaining option, is discussed in Section 3.7.8). When selected, each menu option brings up a dialogue box prompting the user for further information. There is always the option to cancel an operation - i.e. *not* quit from the program or *not* start working on a new design.

3.7.6.1 New

Selecting this menu option brings up a dialogue box containing editors for the design's name (initially "Untitled") and initial dimensions (500 x 500 pixels) and two light buttons *New* (=go-ahead) and *Cancel*. The user may start working on a new design by entering the desired values into the editors and clicking on *New*.

The window which is generated by *New* is known as the *root window* of a design. It may be moved about the main editor window and resized by dragging its borders. Clicking inside the window with the middle mouse button re-displays the properties dialogue, allowing sizes to specified explicitly and the window's title changed.

3.7.6.2 Close

An interface design can be closed by selecting *Close* from the *File* menu. This brings up a dialogue asking which design to close. The required design is selected by clicking on it and then clicking on *Close*.

3.7.6.3 Load and Save

Interface designs can be loaded and saved to both the persistent store and UNIX files. Selecting *Save* brings up a dialogue similar to that for *Close*, giving a menu of designs currently being edited and asking which is to be saved.

Having specified this a second dialogue appears asking whether the design is to be saved to the persistent store or to a UNIX file and the pathname to be used.

For UNIX files the pathname is relative to the current directory. For a store, the pathname is the path from the root of the store (not included), e.g.:

```
User/InterfaceDesigns/database
```

would mean that the interface design was to be saved in the store under the name *database* within the environment *InterfaceDesigns* within the environment *User* in the root environment.

A saved design can be loaded back into the editor with the option *Load* from the *File* menu.

3.7.6.4 Quit

Selecting the menu option *Quit* will undisplay the interface editor window.

3.7.7 Interactors Menu

Having created the overall window for an interface using the *New* option in the *File* menu, the next stage is to lay out the interactors which will make up the interface.

3.7.7.1 Layout Mode

In *Layout* mode interactors are represented by windows of the appropriate size. Windows can be resized directly, either by the properties dialogue (see Section 3.7.7.2) or by dragging out the window to the required size. The size of windows for some other interactors, e.g. menus, depends rather on the component parts of the interactor, e.g. how many entries a menu has and how big a window would be required to display them all.

The interactor types whose windows may be resized are: windows, window managers, sliders, simple and generic dialogues and text editors. All others are sized according to the properties specified by the user in the associated properties dialogue.

The type of each interactor is displayed textually within its window if the window is large enough. Thus a window interactor would have "WINDOW" written inside it and a button interactor would contain the text "BUTTON". If the dimensions of the window are such that this identifying tag would not fit in horizontally but would fit in vertically if rotated 90 degrees anti-clockwise, then it is displayed in this way.

3.7.7.2 Creating an Interactor

To create an instance of an interactor, select the corresponding entry from the *Interactors* menu. This brings up a dialogue window allowing the properties of the interactor to be specified. Edit these as appropriate then click on *Create* to generate a window representing the interactor. This window will be placed in the centre of the current window manager and can be moved to the required position. Clicking with the middle mouse button on this window re-displays the properties dialogue, allowing changes to be made to the interactor after its creation.

3.7.7.3 Border Menus

As well as handling move and resize operations, the borders of interactor windows contain menus which are summoned by clicking with the right mouse button on the border area. The menus contain three options: *front*, *back* and *delete*. The first two give the user control over the ordering of overlapping interactor windows like the border menus of root windows of designs. The last option allows an interactor, once created, to be deleted. Selecting this brings up a dialogue window asking for confirmation that the interactor is to be deleted. Only if confirmation is given will the interactor be deleted.

3.7.8 Code Generation

Selecting *Generate* from the *File* menu brings up a dialogue containing a menu of designs currently being edited and asking which is to have code generated for it. Click on the required design, then on the *Generate* button to start the generation. There will be a short delay while the code is generated. When this is complete, an editor containing the generated code will be displayed. The code can then be edited, saved, evaluated, etc, as required.

The structure of the generated code is as follows (comments in italics):

```
project PS() as root onto env :
use root with …
       binding to locations in the store for use later, i.e. generators, failvalues etc.
in

proc( → Window )
begin
       procedure to generate a window implementing the interface design
end
default : proc( → Window ) ; windowFailValue
```

Figure 3.4: Structure of generated code

The action procedures for the interactors may be filled in to give the application-specific behaviour of the interface. The appropriate sections of the code are highlighted with comments.

4 The *Library* Environment

This section describes the contents of the standard *Library* environment. Each environment within *Library* is described in a sub-section of the same name, thus for example *Section 4.1 Arithmetic* describes the contents of the environment *Arithmetic* within *Library*.

Most of the types used here are defined in Section 6. They are available in the declaration set *userTypes*. The types *Binding*, *TypeRep* and *TypeDescriptor* are value-dependent types and their use is described in Section 4.18.

4.1 *Arithmetic*

---

*abs : **proc(int → int)***

This procedure returns the absolute value of the parameter. If the parameter is equal to

-maxint - 1

where the value of *maxint* is described later in this section, the procedure calls *unaryInt* in the error environment described in Section 5.1, passing it the parameter. In this case, the result obtained from the call of *unaryInt* is returned as the result of *abs*.

---

*atan : **proc(real → real)***

This procedure returns the arctangent of the parameter *x* (given in radians) where:

$$-\frac{\pi}{2} < \text{atan}(x) < \frac{\pi}{2}$$

On an error, this procedure calls *unaryReal* in the error environment, passing it the parameter. In this case, the result obtained from the call of *unaryReal* is returned as the result of *atan*.

---

*bitwiseAnd : **proc(int, int → int)***

This procedure returns the logical (bitwise) **and** of the parameters.

---

*bitwiseOr : **proc(int, int → int)***

This procedure returns the logical (bitwise) **or** of the parameters.

---

*bitwiseNot : **proc(int → int)***

This procedure returns the logical (bitwise) **not** of the parameter.

---

*cos : **proc(real → real)***

This procedure returns the cosine of the parameter (given in radians). On an error, this procedure calls *unaryReal* in the error environment, passing it the parameter. In this case, the result obtained from the call of *unaryReal* is returned as the result of *cos*.

---

*epsilon : **real***

This is the largest value, ε, such that 1.0 + ε = 1.0 in the implementation.

---

*exp : **proc(real → real)***

This procedure returns *e* raised to the power of the parameter. On an error, this procedure calls *unaryReal* in the error environment, passing it the parameter. In this case, the result obtained from the call of *unaryReal* is returned as the result of *exp*.

---

*float : **proc(int → real)***

This procedure returns the parameter expressed as a real number.

---

*ln : **proc(real → real)***

This procedure returns the logarithm to the base *e* of the parameter. If the parameter is not greater than zero, this procedure calls *unaryReal* in the error environment, passing it the parameter. In this case, the result obtained from the call of *unaryReal* is returned as the result of *ln*.

---

*maxint : **int***

This is the maximum integer possible in the implementation.

---

*maxreal : **real***

This is the maximum real possible in the implementation.

---

*pi : **real***

This is the value of π in the implementation.

---

*rabs : **proc(real → real)***

This procedure returns the absolute value of the parameter.

shiftLeft : ***proc(int, int → int)***

This procedure returns the value obtained by performing a bitwise shift left on the first parameter by the number of places given by the second parameter. Zeros are brought in at the low order end.

---

shiftRight : ***proc(int, int → int)***

This procedure returns the value obtained by performing a bitwise shift right on the first parameter by the number of places given by the second parameter. Zeros are brought in at the high order end.

---

sin : ***proc(real → real)***

This procedure returns the sine of the parameter (given in radians). On an error, this procedure calls *unaryReal* in the error environment, passing it the parameter. In this case, the result obtained from the call of *unaryReal* is returned as the result of *sin*.

---

sqrt : ***proc(real → real)***

This procedure returns the positive square root of the parameter. If the parameter is negative the procedure calls *unaryReal* in the error environment, passing it the parameter. In this case, the result obtained from the call of *unaryInt* is returned as the result of *sqrt*.

---

truncate : ***proc(real → int)***

This procedure returns the integer *i* such that for the parameter *x*,

$|i| \leq |x| \leq |i| + 1$ where i * x ≥ 0.

On an error, this procedure calls *truncate* in the error environment, passing it the parameter. In this case, the result obtained from the call of *truncate* is returned as the result.

4.2 *Browser*

---

graphicalBrowserGen : ***proc****(WindowManager →* ***proc****(Binding))*

This procedure creates a browser procedure that displays output on the given window manager. The browser procedure displays a graphical representation of its parameter.

---

textualBrowserGen : ***proc(proc(string)*** *→* ***proc****(Binding))*

This procedure creates a browser procedure that displays textual output using the given procedure. The browser procedure writes out a textual representation of its parameter.

4.3 *Compiler*

---

*compileHyperSource : **proc**(HyperText[Binding] → CompilationResult[TypeDescriptor])*

This procedure takes a hyper-program source representation and attempts to compile it. The result is a variant with the following branches:

---

*voidResult : **proc**()*

This branch is obtained when a void sequence [MBC+94a] is compiled successfully. Calling the procedure causes the sequence to be executed.

---

*nonVoidResult : **proc**(→ **any**)*

This branch is obtained when a non-void sequence is compiled successfully. Calling the procedure causes the sequence to be executed and the result returned, injected into *any*.

---

typeExpression : TypeDescriptor

This branch is obtained when a type expression is compiled successfully. The value is a protected representation of a type and optional constructor information.

---

*error : *CompilationError*

This branch is obtained when the compilation fails. The vector contains an element for each reported compilation error. Each element is a structure containing the following:

errorRegion : CodeRegion

This contains the character offsets of the beginning and end of the error region. Note that this is the region in which the error was first detected; the erroneous code may lie before this.

errorLine : CodeRegion

This contains the character offsets of the beginning and end of the line containing the error region.

*lineNumber : **int***

This is the number of the line containing the error region.

*errorMessage : **string***

This is a message describing the nature of the error.

---

*compileHyperSourceWith : **proc**(HyperText[Binding], ***string** →*

CompilationResult[TypeDescriptor])

This procedure performs the same function as *compileHyperSource*. The additional vector parameter contains the names of declaration sets against which the source is compiled. The declaration set corresponding to the vector element with the lowest index forms the innermost additional scope.

---

*compileString : **proc**(**string** → CompilationResult[TypeDescriptor])*

This procedure takes a string program representation and attempts to compile it. The result has the same form as that of *compileHyperSource*. Calling this procedure has the same effect as calling *genericCompile* with the environment produced by calling *stringInput* with the string as parameter.

---

*compileStringWith : **proc**(**string**, ***string** → CompilationResult[TypeDescriptor])*

This procedure performs the same function as *compileString*. The additional vector parameter contains the names of declaration sets against which the source is compiled. The declaration set corresponding to the vector element with the lowest index forms the innermost additional scope.

---

*compileTypeDefinitions : **proc**(HyperText[Binding] →*
TypeCompilationResult[TypeDescriptor])

This procedure takes a hyper-program source representation containing type definitions and attempts to compile it. The result is a variant which takes one of the following branches:

---

*typeDefinitions : Table[**string**, TypeDescriptor]*

This branch is obtained when the source is compiled successfully. The table contains those type definitions in scope at the end of the program.

---

*error : *CompilationError*

This branch gives compiler error messages and is obtained when the compilation fails.

---

*fileInput : **proc**(**file** → **env**)*

This procedure takes a file descriptor and returns an environment containing lexical analysis procedures to operate over that file.

---

*genericCompile : **proc**(**env** → CompilationResult[TypeDescriptor])*

This procedure takes an environment containing lexical analysis procedures operating over a source representation and attempts to compile the source. The result has the same form as that of *compileHyperSource*. A compiler error message is obtained if the environment does not contain the following procedures:

*eoi : **proc(→ bool)***

This procedure should return **true** iff the end of the source has been reached.

---

*read : **proc(→ string)***

This procedure should return the next character from the source and advance to the following character.

---

*readName : **proc(string → string)***

This procedure should read an identifier name from the source and return it appended to the parameter.

---

*resetLex : **proc()***

This procedure should reset the current lexical position to the beginning of the source.

---

*sourceOffset : **proc(→ int)***

This procedure should return the current character offset into the source.

---

*positionInfo : **proc(int → int)***

This procedure should return, for a given offset into the source, a vector with a lower bound of 1 and the following values in the first three elements: the character offset of the start of the line containing the given offset, the character offset of the end of the line containing the given offset, and the number of the line containing the given offset. The offset parameter specifies a character in the source, with 1 corresponding to the first character. The returned line start and end offsets should correspond to the first and last characters of the line excluding newline characters. The line numbering should start at 1.

---

*sourceFragment : **proc(int, int → string)***

This procedure should return the fragment of the source between and including the given character offsets.

---

*getDeclarationSet : **proc(string** → Optional[Table[**string**, Binding]])*

This procedure returns a table operating on the declaration set with the given name, if it exists. This table can then be used to enter, look up, remove and scan bindings in the declaration set.

---

*newDeclarationSet : **proc(string)***

This procedure creates a new declaration set with the given name, unless one already exists in which case it has no effect.

---

*removeDeclarationSet : **proc(string)***

This procedure removes the declaration set with the given name, unless no such declaration set exists in which case it has no effect.

---

*scanDeclarationSets : **proc(proc(string → bool))***

This procedure calls the given procedure repeatedly, passing it the name of each declaration set, until it has been called for all declaration sets or it returns **false**. The declaration sets are scanned in increasing name order.

---

*stringInput : **proc(string → env)***

This procedure takes a string and returns an environment containing lexical analysis procedures to operate over that string.

4.4 *Concurrency*

---

*semaphoreGen : **proc(int** → Semaphore)*

This procedure takes an initial value for a semaphore and returns a structure containing procedures to operate on the semaphore. If the parameter is negative an initial value of zero is used. The operations on the semaphore are defined as in [SPG91] p. 153:

---

*wait : **proc()***

The value of the semaphore is decremented. If the new value is less than zero then the current thread is suspended and its dependency on the semaphore is recorded.

---

*signal : **proc()***

The value of the semaphore is incremented. If the new value is less than or equal to zero, one of the threads suspended on the semaphore is selected and made runnable.

---

threadPackage : ThreadPack

This abstract datatype instance contains procedures to operate on threads. Its closure contains a set of threads, each of which may be runnable or suspended. At any time while the Napier88 system is active, one or more of the runnable threads are executing. The programmer may manipulate threads as witnesses of the abstract datatype. Denoting the witness type as *Thread*, the operations are:

---

*start : **proc**(**proc**() → Thread)*

This procedure creates a new thread to execute the given void procedure, adds the thread to the set of threads, marks the thread as runnable, and returns an identifier for the thread.

---

*getCurrentThread : **proc**(→ Thread)*

This procedure returns the identifier of the thread executing it.

---

*getAllThreads : **proc**(→ *Thread)*

This procedure returns a vector containing identifiers for all the current members of the set of threads.

---

*kill : **proc**(Thread)*

This procedure removes the thread denoted by the given identifier from the set of threads. If the thread is currently executing it is terminated. If no runnable threads remain the Napier88 system terminates.

---

*restart : **proc**(Thread)*

This procedure marks the thread denoted by the given identifier as runnable. If the thread is currently executing the procedure has no effect.

---

*suspend : **proc**(Thread)*

This procedure marks the thread denoted by the given identifier as suspended. If the thread is currently executing it is suspended immediately. If no runnable threads remain the Napier88 system terminates.

---

*live : **proc**(Thread → **bool**)*

This procedure returns **true** iff the given thread identifier denotes a thread which has not yet terminated.

---

*getParent : **proc**(Thread → Optional[Thread])*

This procedure returns the identifier of the thread from which the given thread was started, if it has not yet terminated. The *absent* branch is returned if the parent has terminated or if the given thread has no parent.

---

*suspendUnlock : **proc**(**string**, Thread)*

This procedure is for system use only and is password protected.

4.5 *Device*

---

getScreen : ***proc(file → image)***

If the parameter is a raster device, this procedure returns the image associated with that device. If the file is not a raster device, a call is made to the procedure *getScreen* in the error environment with the parameter supplied to the original call. In this case, the result obtained from the call of the error procedure is returned as the result.

---

locator : ***proc(file, *int)***

If the file parameter is a mouse or tablet device, this procedure copies the locator information into the vector parameter.

If the file is not a mouse or tablet device, a call is made to the procedure *locator* in the error environment with the parameters supplied to the original call.

The elements of the vector are filled in as follows:

1	if the file is a tablet, its X dimension, otherwise 0,
2	if the file is a tablet, its Y dimension, otherwise 0,
3	the locator X position,
4	the locator Y position,
5	a date stamp,
6	the state of button 1, representing down as 1 and up as 0,
7...n	the state of button *i* - 5 where *i* is the vector index.

If the vector has more elements than the information available, the extra elements are ignored; if the vector has too few elements, only the elements present are set. If the file is a tablet device or a mouse device associated with an X-window, the X and Y positions are absolute. Otherwise the X and Y positions are relative to the those of the last call of *locator*.

---

colourMap : ***proc(file, pixel, int)***

If the file parameter is a raster device, this procedure sets the colour map entry for the pixel parameter to the integer parameter. The interpretation of the integer is device dependent.

If the file is not a raster device, a call is made to the procedure *colourMap* in the error environment with the parameters supplied to the original call.

---

colourOf : ***proc(file, pixel → int)***

If the file parameter is a raster device, this procedure returns the colour map entry associated with the given pixel for that device.

If the file is not a raster device, a call is made to the procedure *colourOf* in the error environment with the parameters supplied to the original call. In this case, the result obtained from the call of the error procedure is returned as the result.

---

*getCursor : **proc(file → image)***

If the parameter is a raster device, this procedure returns the cursor associated with that device.

If the file is not a raster device, a call is made to the procedure *getCursor* in the error environment with the parameters supplied to the original call. In this case, the result obtained from the call of the error procedure is returned as the result.

---

*setCursor : **proc(file, image)***

If the file parameter is a raster device, this procedure sets the cursor to be a copy of the image parameter.

If the file is not a raster device, a call is made to the procedure *setCursor* in the error environment with the parameters supplied to the original call.

---

*getCursorInfo : **proc(file, *int)***

If the file parameter is a raster device, this procedure copies the cursor information for that device into the vector parameter.

If the file is not a raster device, a call is made to the procedure *getCursorInfo* in the error environment with the parameters supplied to the original call.

The elements of the vector are filled in as follows:

1 the cursor's X position,
2 the cursor's Y position,
3 the rasterop rule used to display the cursor.

The interpretation of the rasterop rule may be found in the description of *rasterOp* in Section 4.12.2.

If the vector has more than three elements, the extra elements are ignored. If the vector has less than three elements, only the elements present are filled in.

---

*setCursorInfo : **proc(file, *int)***

If the file parameter is a raster device, this procedure alters the cursor information for that device according to the contents of the vector parameter.

If the file is not a raster device, a call is made to the procedure *setCursorInfo* in the error environment with the parameters supplied to the original call.

The elements of the vector are interpreted as follows:

1 specifies the cursor's X position,
2 specifies the cursor's Y position,
3 specifies the rasterop rule used to display the cursor.

The interpretation of the rasterop rule may be found in the description of *rasterOp* in Section 4.12.2.

If the vector has more than three elements, the extra elements are ignored; if the vector has less than three elements, only the elements present are set.

4.6 *Distribution*

---

remoteStoreTable : Table[***string****, RemoteStore]*

This table contains mappings from symbolic remote store names to actual locations of stores. The symbolic names are strings, while the store location structures each contain the name of a remote host, the pathname of a store at that host, a user name and a password. The host name may be specified as a local machine name or a full IP host name. The user name and password may be empty strings. For example:

```
remoteStoreTable( enter )(
        "panda",
        RemoteStore( "panda", "/pstore2/demoStore", "", "" ) )

remoteStoreTable( enter )(
        "aRemoteStore",
        RemoteStore( "mcname.somewhere.edu", "/napier/store", "john", "napier" ) )
```

---

scan : ***proc****(RemoteStore,* ***string*** *→ RemoteResult[StoreScan[TypeRep]])*

This procedure takes a remote store description and attempts to connect to that store and return information about the contents of the store. If the store description is not valid the result is a string describing the error. Otherwise the result depends on whether the store contains an environment at the root of persistence.

If the store contains an environment the string parameter is interpreted as the pathname of an environment accessible from the root environment, and the result is a list of structures containing one element for each of the bindings present in the remote environment at the time of the scan. Each element contains the name of the binding as a string and a representation of the type of the binding. The pathname is given relative to the top level environment and should consist of an initial slash followed by environment names separated by slashes, for example:

```
"/Library/Distribution"      ! Distribution contained in Library contained in top level.
"/"                          ! Top level environment.
```

If the pathname is not well formed the result is a string describing the error.

If the store does not contain an environment at the root of persistence, the string parameter is ignored and the result is a representation of the type of the root of persistence.

The operation of this procedure depends on whether a Napier88 process is currently active in the remote store. If so the procedure attempts to connect with the remote process at the socket level and any user name or password provided with the remote store description is ignored. If this attempt fails, or if no process is active in the remote store, the procedure attempts to copy the remote store to the local machine in order to scan it. In this case the user name and password, if present, may be used in the attempt to connect to the remote machine.

---

copyValue : ***proc****(RemoteStore,* ***string*** *→ RemoteResult[* ***any*** *])*

This procedure takes a remote store description and attempts to copy a value from it. If the store contains an environment at the root of persistence the string parameter is interpreted as a

pathname from the root environment in the same way as for *scan* above. In this case the result is a copy of the remote binding injected into *any*. If the store does not contain an environment, the pathname is not well formed or no binding with the given name is present, the result is a string describing the error.

User name and password information is used in the same way as for *scan*.

---

*copyStore : **proc**(RemoteStore → RemoteResult[**any**])*

This procedure takes a remote store description and attempts to make a deep copy of its contents. If the attempt fails the result is a string describing the error, otherwise the result is a copy of the store contents injected into *any*.

User name and password information is used in the same way as for *scan*.

---

*createStore : **proc**(RemoteStore, **any** → RemoteResult[**null**])*

This procedure takes a remote store description and a value injected into *any*, and attempts to create a new store containing that value. If the remote store already exists or the attempt to create the store fails for some other reason, the result is a string describing the error. Otherwise the result is *nil*.

User name and password information is used in the same way as for *scan*.

---

*setListener : **proc**(**bool**)*

This procedure turns listening in the local store on if the parameter is **true** and off otherwise. Listening involves monitoring the network for incoming connection attempts from other stores. If it is turned off no other store will be able to connect to the local store. The performance of threads in the local store will however be increased.

4.7 *Environment*

---

*environment : **proc**(→ **env**)*

This procedure creates a new empty environment.

---

*scan : **proc**(**env**, **proc**(**string**, TypeRep, **bool**))*

This procedure calls the given procedure once for every binding in the given environment, in alphabetical order of binding name. Each invocation of the procedure is passed the name of the identifier in the binding, a representation of its type and a boolean to indicate whether or not the location is constant.

4.8 *Event*

The Napier88 system recognises a small range of asynchronous events. These are a hangup signal, an interrupt signal, a quit signal and a timer interrupt. On completion of a particular event procedure, the procedure will return to the running program.

The *Event* environment contains the procedures that are called when one of these events is detected by the system. These procedures are variables and the user may change them by assignment. The default procedures are described below.

The UNIX signals referred to may be found in §3 of the UNIX Manual under Signal.

---

hangup : ***proc()***

This procedure is called if the Napier88 system receives a UNIX SIGHUP signal. By default, this procedure stops the Napier88 system.

---

interrupt : ***proc()***

This procedure is called if the Napier88 system receives a UNIX SIGINT signal. By default, this procedure does nothing.

---

quit : ***proc()***

This procedure is called if the Napier88 system receives a UNIX SIGQUIT signal. By default, this procedure stops the Napier88 system.

---

timer : ***proc ()***

This procedure is called 30 times per second. By default, this procedure does nothing.

4.9 *FailValues*

The *FailValues* environment contains dummy instances of some commonly used types. The types are defined in Section 5.

applicationFailValue :	Application
bindingFailValue :	Binding
bindingEditorFailValue :	Editor[Binding]
borderStyleFailValue :	BorderStyle
buttonPackFailValue :	ButtonPack
choicePackFailValue :	ChoicePack
displayInfoFailValue :	DisplayInfo
envFailValue :	**env**
fontFailValue :	Font
fontPackFailValue :	FontPack
hyperProgramPackFailValue :	EditorPack[Binding]
hyperSourceFailValue :	HyperText[Binding]
iconManagerFailValue :	IconManager
indexFailValue :	Index
intVectorFailValue :	***int**
levelFailValue :	Level
limitFailValue :	Limit
menuPackFailValue :	MenuPack
posFailValue :	Pos
rectFailValue :	Rect
sizeFailValue :	Size
sliderPackFailValue :	SliderPack

soundFailValue : *__int__
stringVectorFailValue : *__string__
windowFailValue : Window
windowManagerFailValue : WindowManager
windowStateFailValue : WindowState

4.10 *Font*

The *Font* environment contains the following instances of type *FontPack*:

					cmrB14		
					cmrR14		
	courB10		courB12		courB14		
	courR10		courR12		courR14		
							gallantR19
			screenB12		screenB14		
screenR7		screenR11	screenR12	screenR13			
	serifR10	serifR11	serifR12		serifR14		

Table 4.1: Font names

Each instance is a structure with the following fields:

---

font : Font

This structure contains *characters*, a vector of images; *fontHeight*, the height of the characters in pixels; *descender*, the distance from the bottom of a character to the base line; and *info*, a string describing the font.

---

stringToTile : ***proc(string → image)***

This procedure returns a new image onto which the characters of the given string have been copied.

---

charToTile : ***proc(string → image)***

This procedure returns the image corresponding to the first character of the given string. This may be used as an optimisation of *stringToTile* in some cases.

The widths of characters in a font may vary, but the programmer may examine these by taking the *x* dimension of the appropriate image. For example:

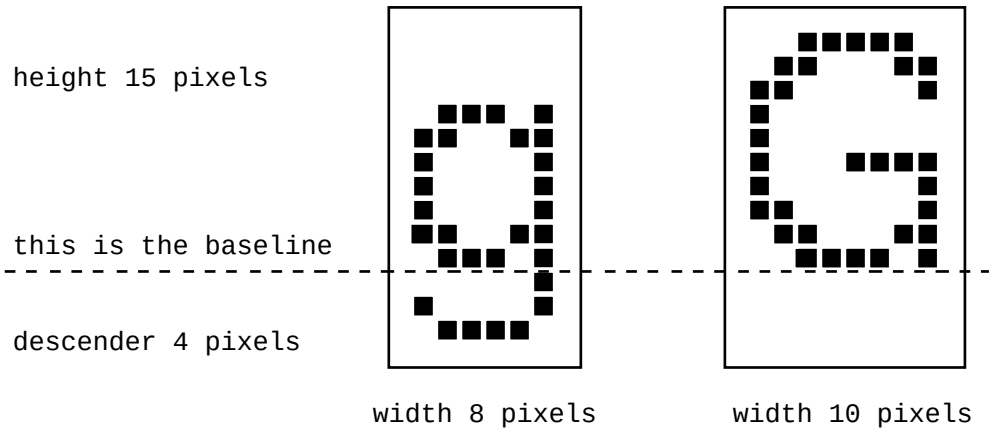


Figure 4.1: Character dimensions

4.11 *Format*

---

*eformat : **proc(real, int, int → string)***

This procedure returns a string representation of the real parameter, with an exponent. The first integer parameter gives the required number of digits before the decimal point and the second the number of digits after the decimal point. If either integer parameter is negative, a call is made to the procedure *eformat* in the error environment with the parameters supplied to the original call. In this case, the result obtained from the call of the error procedure is returned as the result.

---

*fformat : **proc(real, int, int → string)***

This procedure returns a string representation of the real parameter. The first integer parameter gives the required number of digits before the decimal point and the second the number of digits after the decimal point. If the first integer is too small to represent the real number, or the second integer is negative, a call is made to the procedure *fformat* in the error environment with the parameters supplied to the original call. In this case, the result obtained from the call of the error procedure is returned as the result.

---

*gformat : **proc(real → string)***

This procedure returns a string representation of the real parameter in the most suitable format.

---

*iformat : **proc(int → string)***

This procedure returns a string representation of the integer.

4.12 *Graphics*

4.12.1 *Outline*

---

makeDrawFunction : ***proc(string*** *→ drawFunction)*

This procedure is for use with Outline graphics [Mor82, MBB+86]. It takes a string parameter describing a device type and returns a procedure to display Outline pictures on that device. The devices supported are:

"image"	Napier88 raster image
"g6320"	colour plotter

If the parameter is not one of the strings listed above the null branch of the variant is returned. For the parameter "image" the result is of type:

proc(**image**, **pic**, **real**, **real**, **real**, **real**)

otherwise the result is of type:

proc(**file**, **pic**, **real**, **real**, **real**, **real**)

In either case the procedure returned takes a picture and a bounding rectangle in the infinite two dimensional real space over which all pictures are defined. The real parameters give the minimum *x*, maximum *x*, minimum *y* and maximum *y* bounds respectively. The picture is clipped to the area of the bounding rectangle. The rectangle is then scaled and shifted to fit the area of the device on which it is drawn.

If the *x* parameters are equal or the *y* parameters are equal then the bounding box has zero size and a call is made to the *Draw* procedure in the graphical errors environment.

If the picture being drawn contains a text statement whose end points are coincident, a call is made to the *Text* procedure in the graphical errors environment. The result returned by the *Text* procedure is used to replace the erroneous text statement.

The mapping of a picture onto a device is performed using real arithmetic which, in certain circumstances, may result in arithmetic errors. If any arithmetic errors do occur the appropriate procedure in the arithmetical errors environment is called.

4.12.2 *Raster*

This environment contains procedures for use with raster graphics [MBD+86].

---

xDim : ***proc(image*** *→* ***int)***

This procedure returns the X dimension of the image.

---

yDim : ***proc(image*** *→* ***int)***

This procedure returns the Y dimension of the image.

---

zDim : ***proc(image → int)***

This procedure returns the number of planes in the image.

---

pixelDepth : ***proc(pixel → int)***

This procedure returns the number of planes in the pixel.

---

rasterOp : ***proc(image, image, int)***

This procedure performs a raster operation from the first image *S* onto the second image *D* using a rule given by dividing the integer parameter by 16 and interpreting the remainder as follows:

0	S **and** ~S	8	S **and** D
1	~ (S **or** D)	9	~S **xor** D
2	~S **and** D	10	D
3	~S	11	~S **or** D
4	S **and** ~D	12	S
5	~D	13	S **or** ~D
6	S **xor** D	14	S **or** D
7	~ (S **and** D)	15	S **or** ~S

Table 4.2: Raster rules

where **on** maps to **true** and **off** maps to **false**.

---

line : ***proc(image, int, int, int, int, pixel, int)***

This procedure draws a line on the image parameter. The *x* and *y* coordinates of the first end point are given by the first and second integer parameters respectively. The *x* and *y* coordinates of the second end point are given by the third and fourth integer parameters respectively. The line is drawn using the pixel parameter which is combined with the pixels of the image using the raster rule given by the last parameter. The interpretation of the raster rule is the same as for *rasterOp*.

---

getPixel : ***proc(image, int, int → pixel)***

This procedure returns the pixel at the given position in the image. The first integer parameter gives the *x* coordinate and the second integer parameter the *y* coordinate. If the position lies outside the image a call is made to the procedure *getPixel* in the error environment. In this case, the result obtained from the call of the error procedure is returned as the result.

*setPixel : **proc(image, int, int, pixel)***

This procedure sets the pixel at the given position in the image. The first integer parameter gives the *x* coordinate and the second integer parameter the *y* coordinate. If the position lies outside the image a call is made to the procedure *setPixel* in the error environment.

4.13 *InteractiveEnvironment*

*startProgrammingEnv : **proc()***

This procedure starts the interactive programming environment at its previously stabilised state. The procedure attempts to connect to the X-server indicated by the UNIX environment variable DISPLAY and to create a window in which to run the programming environment. If the display is opened successfully, hyper-programming and browser windows are displayed as described in Section 2.

4.14 *InterfaceEditor*

*interfaceEditorGen : **proc(→ Window)***

This procedure returns a window containing a user interface editor as described in Section 3.7.

4.15 *IO*

*stdOut : **file***

This is a file variable that is initially connected to the UNIX control terminal for the Napier88 system.

*writeByte : **proc(int)***

This procedure computes the bitwise **and** of the integer with 255 and writes the result as a byte to the file *stdOut*. If an error occurs a call is made to the procedure *writeByte* in the error environment.

*writeString : **proc(string)***

This procedure writes the string to the file *stdOut*. If an error occurs a call is made to the procedure *writeString* in the error environment.

*writeBool : **proc(bool)***

This procedure writes the boolean to the file *stdOut*. If an error occurs a call is made to the procedure *writeBool* in the error environment.

---

*writeInt : **proc(int)***

This procedure writes the integer to the file *stdOut*. If an error occurs a call is made to the procedure *writeInt* in the error environment.

---

*writeReal : **proc(real)***

This procedure writes the real to the file *stdOut*. If an error occurs a call is made to the procedure *writeReal* in the error environment.

---

*integerWidth : **int***

Integers written out using *writeInt* are displayed, left justified, in this number of characters. If the number does not fit within this space, the exact number of characters is used. *integerWidth* is a variable with an initial value of 12.

---

*realWidth : **int***

Reals written out using *writeReal* are displayed, left justified, in this number of characters. If the number does not fit within this space, the exact number of characters is used. *realWidth* is a variable with an initial value of 14.

---

*spaceWidth : **int***

spaceWidth spaces are written out after any integer or real number written using *writeInt* or *writeReal*. *spaceWidth* is a variable with an initial value of 2.

---

*makeWriteEnv : **proc(file → env)***

This procedure creates an environment that contains the procedures *writeByte*, *writeString*, *writeBool*, *writeInt* and *writeReal*, each of which operates on the given file rather than the file *stdOut*. Each procedure may call the error procedures described above. The environment also contains the variables *integerWidth*, *realWidth* and *spaceWidth*, to control the operation of *writeInt* and *writeReal* on the file. The initial values of the three variables are 12, 14 and 2 respectively.

---

*stdIn : **file***

This is a file variable that is initially connected to the UNIX control terminal for the Napier88 system.

---

*endOfInput : **proc(→ bool)***

This procedure reads one byte as an integer from the file *stdIn*. If the read is successful, **false** is returned. If an I/O error occurs the procedure calls the *endOfInputIOE* procedure in the error environment. If the end of input is encountered the procedure returns **true**. The procedure attempts to make the byte read available to the next *endOfInput*, *readByte*, *readChar*, *peekByte*, *peekChar*, *readString*, *readLine*, *readBool*, *readInt* or *readReal* operation. If the byte cannot be made available, a call is made to the *endOfInputUnread*

procedure in the error environment. The result obtained from any of the error procedures is returned as the result of *endOfInput*.

---

inputPending : ***proc(→ bool)***

This procedure returns **true** iff there is input available to be read from the file *stdIn*.

---

readByte : ***proc(→ int)***

This procedure reads one byte as an integer from the file *stdIn*. If an I/O error occurs the procedure calls the *readByteIOE* procedure in the error environment. If the end of input is encountered the procedure calls the *readByteEOI* procedure in the error environment. The result obtained from either of the error procedures is returned as the result of *readByte*.

---

readChar : ***proc(→ string)***

This procedure reads one character from the file *stdIn*. If an I/O error occurs the procedure calls the *readCharIOE* procedure in the error environment. If the end of input is encountered the procedure calls the *readCharEOI* procedure in the error environment. The result obtained from either of the error procedures is returned as the result of *readChar*.

---

peekByte : ***proc(→ int)***

This procedure reads one byte as an integer from the file *stdIn*. If an I/O error occurs the procedure calls the *peekByteIOE* procedure in the error environment. If the end of input is encountered the procedure calls the *peekByteEOI* procedure in the error environment. The procedure attempts to make the byte read available to the next *endOfInput*, *readByte*, *readChar*, *peekByte*, *peekChar*, *readString*, *readLine*, *readBool*, *readInt* or *readReal* operation. If the byte cannot be made available, a call is made to the *peekByteUnread* procedure in the error environment. The result obtained from any of the error procedures is returned as the result of *peekByte*.

---

peekChar : ***proc(→ string)***

This procedure reads one character from the file *stdIn*. If an I/O error occurs the procedure calls the *peekCharIOE* procedure in the error environment. If the end of input is encountered the procedure calls the *peekCharEOI* procedure in the error environment. The procedure attempts to make the character read available to the next *endOfInput*, *readByte*, *readChar*, *peekByte*, *peekChar*, *readString*, *readLine*, *readBool*, *readInt* or *readReal* operation. If the character cannot be made available, a call is made to the *peekByteUnread* procedure in the error environment. The result obtained from any of the error procedures is returned as the result of *peekChar*.

---

readString : ***proc(→ string)***

This procedure reads a string literal (a string in quotes) from the file *stdIn*. The layout characters " ", "'t" and "'n" are ignored.

If the first character after any layout characters is not a double quote the procedure calls the *readStringBadChar* procedure in the error environment. The erroneous character will have been read. If an I/O error occurs the procedure calls the *readStringIOE* procedure in the error

environment. If the end of input is encountered the procedure calls the *readStringEOI* procedure in the error environment. The result obtained from any of the error procedures is returned as the result of *readString*.

---

readLine : ***proc****(→ **string**)*

This procedure reads characters from the file *stdIn* up to and including a newline character. It concatenates the characters and returns them as a string without the newline character. If the end of input is encountered during this operation the procedure calls the *readLineEOI* procedure in the error environment. If an I/O error occurs the procedure calls the *readLineIOE* procedure in the error environment. The result obtained from either of the error procedures is returned as the result of *readLine*.

---

readBool : ***proc****(→ **bool**)*

This procedure reads one boolean from the file *stdIn*. The layout characters " ", "'t" and "'n" are ignored. If the characters after any layout characters do not form a boolean the procedure calls the *readBoolBadChar* procedure in the error environment. The characters up to and including the first erroneous character will have been read. If an I/O error occurs, the procedure *readBool* calls the *readBoolIOE* procedure in the error environment. If the end of input is encountered the procedure calls the *readBoolEOI* procedure in the error environment. The result obtained from any of the error procedures is returned as the result of *readBool*.

---

readInt : ***proc****(→ **int**)*

This procedure reads one integer from the file *stdIn*. The layout characters " ", "'t" and "'n" are ignored. If the first character after any layout characters is not a digit or a sign which is followed by a digit, the procedure calls the *readIntBadChar* procedure in the error environment. The erroneous character will have been read. If the end of input is encountered before the first digit the procedure calls the *readIntEOI* procedure in the error environment. If an I/O error occurs the procedure calls the *readIntIOE* procedure in the error environment.

The procedure reads characters from the file *stdIn* until it has parsed an integer. The parsing may involve reading the first character following the integer. When this occurs the procedure attempts to make the extra character read available to the next *endOfInput*, *readByte*, *readChar*, *peekByte*, *peekChar*, *readString*, *readLine*, *readBool*, *readInt* or *readReal* operation. If the character cannot be made available a call is made to the *readIntUnread* procedure in the error environment.

When an integer has been successfully parsed it is converted into an integer value. If an arithmetic error occurs during the conversion, a call is made to the *readIntOverflow* procedure in the error environment. The result obtained from any of the error procedures is returned as the result of *readInt*.

---

readReal : ***proc****(→ **real**)*

This procedure reads one real from the file *stdIn*. The layout characters " ", "'t" and "'n" are ignored. If the first character after any layout characters is not a digit or a sign which is followed by a digit, the procedure calls the *readRealBadChar* procedure in the error environment. The erroneous character will have been read. If the end of input is encountered before the first digit the procedure calls the *readRealEOI* procedure in the error environment. If an I/O error occurs the procedure calls the *readRealIOE* procedure in the error environment.

This procedure reads characters from the file *stdIn* until it has parsed a real. The parsing may involve reading the first character following the real. When this occurs the procedure attempts to make the extra character read available to the next *endOfInput*, *readByte*, *readChar*, *peekByte*, *peekChar*, *readString*, *readLine*, *readBool*, *readInt* or *readReal* operation. If the character cannot be made available a call is made to the *readRealUnread* procedure in the error environment.

When a real has been successfully parsed it is converted into a real value. If an arithmetic error occurs during the conversion, a call is made to the *readRealOverflow* procedure in the error environment. The result obtained from any of the error procedures is returned as the result of *readReal*.

---

*makeReadEnv : **proc(file → env)***

This procedure creates an environment that contains the procedures *endOfInput*, *inputPending*, *readByte*, *readChar*, *peekByte*, *peekChar*, *readString*, *readLine*, *readBool*, *readInt* and *readReal*, each of which operates on the given file rather than the file *stdIn*. Each of these procedures may call the error procedures described above.

4.15.1 *PrimitiveIO*

The procedures in this environment map the I/O facilities of UNIX onto the Napier88 system.

---

*create : **proc(string, int → file)***

This procedure creates a file with the given name. The integer parameter specifies the decimal value of the (UNIX) file protection bitmap. If the creation fails, **nilfile** is returned.

---

*open: **proc(string, int → file)***

This procedure opens the file with the given name in access mode given by the integer parameter interpreted as follows:

0	read only
1	write only
2	read and write

If the open fails, **nilfile** is returned.

A particular file type and attributes may be specified by prefixing the file name with one of the following prefixes:

“DISK:”, “TTY:”, “STDIN:”, “STDOUT:”, “STDERR:”,
“ACCEPT:”, “CONNECT:”, “SHELL:”, “WINDOW:”

If no recognised prefix is given the host operating system is interrogated after a file is opened/created to determine its type.

Disk file objects are created whenever a file is opened or created in an external file system. The filename prefix for a disk file is "DISK:", for example:

```
open( "DISK:myfile", 2 )
```

Terminal file objects are created whenever a terminal device is opened. The filename prefix is "TTY:". If the filename prefixes "STDIN:", "STDOUT:" or "STDERR:" are specified then file objects are created for the Napier system's standard input, output and error. These files are permanently open and are assumed to be terminal devices. For "STDIN:", "STDOUT:" and "STDERR" the access mode parameter is ignored For example:

```
open( "TTY:/dev/ttyp1", 2 )
open( "STDIN:", 2 )
open( "STDOUT:", 2 )
open( "STDERR:", 2 )
```

A socket file object is created whenever an incoming network connection is accepted or a connection to a remote Napier system is successful. The filename prefixes for a socket are "ACCEPT:", "CONNECT:" and "SHELL:". The access mode parameters are ignored.

"ACCEPT:" is used to accept a connection from any remote Napier system. The remainder of the filename is ignored. If no other Napier system is attempting to connect then **nilfile** is returned. For example:

```
open( "ACCEPT:", 2 )
```

"CONNECT:" is used to connect to a remote Napier system. This is specified by a host identifier, followed by a double colon and the path name of a Napier store directory. The host identifier may be either a local name or full internet address. If the connection attempt fails then **nilfile** is returned. Possible reasons for failure include:

- the host identifier is not a valid address;
- the store directory does not exist or does not contain a valid Napier store; or
- there is no interpreter currently running against the remote store.

For example:

```
open( "CONNECT:panda::/pstore2/demoStore", 2 )
open( "CONNECT:mcname.somewhere.edu::/pstore2/demoStore", 2 )
```

"SHELL:" is used to specify a socket connected to a command line interpreter. The command line interpreter is started when the Napier system is invoked. In a UNIX system the interpreter is a shell.

```
open( "SHELL:", 2 )
```

A window file object is created whenever a raster window is opened. The filename prefix for a window is "WINDOW:". If no window name is given a default window is opened in the host environment. For example, a shell variable DISPLAY may have been set to specify an X display to use. Alternatively it may be possible to access the local frame buffer and use that to simulate a window.

A window filename may include specifications of the *x*, *y* and *z* dimensions of the window as well as its initial *x* and *y* positions. The specifications are encoded by prefixing a number by either "XDIM:", "YDIM:", "ZDIM:", "XPOS:" or "YPOS:" respectively. Each of these attributes is prefixed by a space character to separate them from the rest of the filename. If possible these specifications will be used. If no *z* dimension is specified a default of 1 is assumed. For example:

```
open( "WINDOW: XDIM: 600 YDIM: 600 XPOS: 50 YPOS: 50 ZDIM:8", 2 )
```

---

close : ***proc(file → int)***

This procedure closes the file associated with the given file descriptor. The integer returned is 0 if the operation was successful and -1 otherwise.

---

seek : ***proc(file, int, int → int)***

This procedure sets the position of the next read or write from the given file. The first integer parameter gives the offset in the file relative to the position determined by the second integer parameter as follows:

0	start of file
1	current position
2	end of file

The procedure returns the position in the file if the operation was successful and -1 otherwise.

---

ioctl : ***proc(file, *int, int → int)***

The ioctl commands correspond exactly to those supported by the UNIX *ioctl* system call. The ioctl instruction will not execute the specified command unless it is applicable to a compatible terminal and the vector of integers contains sufficient integer elements to hold the parameters or results of the specified command. The supported commands are:

TIOCSETP	TIOCSETN	TIOCSETC	TIOCSLTC	TIOCSETD
TIOCFLUSH	TIOCSTI	TIOCSPGRP	TIOCLBIS	TIOCLBIC
TIOCEXCL	TIOCNXCL	TIOCHPCL	TIOCSBRK	TIOCCBRK
TIOCSDTR	TIOCCDTR	TIOCSTOP	TIOCSTART	TIOCGETP
TIOCGETC	TIOCGLTC	TIOCGETD	TIOCGPRG	TIOCOUTQ
FIONREAD	FIONBIO			

---

readBytes : ***proc(file, *int, int, int → int)***

This procedure reads bytes from the given file into the vector of integers. The first integer parameter gives the byte offset from the start of the vector's elements. The second integer

parameter gives the maximum number of bytes to be read. The procedure returns the number of bytes read if the operation completes successfully and -1 otherwise. The number of bytes read is not necessarily the maximum possible.

---

writeBytes : ***proc(file, *int, int, int → int)***

This procedure writes bytes to the given file from the vector of integers. The first integer parameter gives the byte offset from the start of the vector's elements. The second integer parameter gives the maximum number of bytes to be written. The procedure returns the number of bytes written if the operation completes successfully and -1 otherwise. The number of bytes written is not necessarily the maximum possible.

---

getByte : ***proc(int, int → int)***

This procedure returns a byte from the word given by the first parameter. The second parameter gives the byte index from the start of the word, 0 indicating the first byte. If an illegal index is specified a call is made to the *getByte* procedure in the error environment.

---

setByte : ***proc(int, int, int → int)***

This procedure returns the integer obtained by replacing a byte in the word given by the first parameter. The second parameter gives the byte index from the start of the word, 0 indicating the first byte. The third parameter gives the byte with which it is to be replaced. If an illegal index is specified a call is made to the *setByte* procedure in the error environment.

---

errorNumber : ***proc(→ int)***

This procedure returns the error number of the last primitive I/O operation executed by the current thread. The error numbers are those returned by the last UNIX I/O operation and are described in intro(2) in the UNIX Manual.

4.16 *Lists*

---

listPackGen : ***proc***[*T*](→ *ListPack*[*T*])

This procedure returns a structure containing procedures to manipulate a list with elements of type *T*. The list implementation maintains a *current position* in the list, represented by an integer specifying the number of list elements before the current position. This may range between 0 and the number of elements in the list. Initially the list is empty and the current position is 0. The procedures are:

---

insert : ***proc***(*T*)

This procedure inserts an element into the list at the current position. The current position now lies after the new element.

---

replace : ***proc****(T)*

This procedure has no effect if the list is empty. Otherwise it replaces the element at the current position with the given element. The current position now lies at the new element.

---

clear : ***proc****()*

This procedure deletes all the elements in the list.

---

delete : ***proc****()*

This procedure has no effect if the current position is equal to the number of list elements i.e. at the end of the list. Otherwise it deletes the element at the current position. The current position remains unchanged.

---

element : ***proc****(→ Optional[T])*

This procedure returns the *absent* branch if the list is empty or the current position is equal to the number of list elements i.e. at the end of the list. Otherwise it returns the element at the current position. The current position remains unchanged.

---

length : ***proc****(→* ***int*** *)*

This procedure returns the number of elements in the list.

---

pos : ***proc****(→* ***int*** *)*

This procedure returns the current position.

---

atEnd : ***proc****(→* ***bool*** *)*

This procedure returns **true** iff the current position lies at the end of the list.

---

go : ***proc****(* ***int*** *)*

This procedure sets the current position to the given value. If the value is less than 0 or greater than the number of list elements the procedure has no effect.

---

goNext : ***proc****()*

This procedure increments the current position by 1. If the current position is already at the end of the list the procedure has no effect.

---

*goPrev : **proc()***

This procedure decrements the current position by 1. If the current position is already at the start of the list the procedure has no effect.

---

*find : **proc(proc(T → bool) → int)***

This procedure scans the elements of the list in order from the start of the list, applying the given procedure to each element, until **true** is obtained or the end of the list is reached. The procedure returns the position of the element for which **true** was obtained, or -1 if there was no such element.

4.17 *People*

This environment contains the following:

al : **image**	Al Dearle, University of Adelaide
carl : **image**	Carl Warren, University of St Andrews
craig : **image**	Craig Baker, University of St Andrews
dave : **image**	Dave Munro, University of St Andrews
dharini : **image**	Dharini Subramaniam, University of St Andrews
fred : **image**	Fred Brown, University of Adelaide
graham : **image**	Graham Kirby, University of St Andrews
john : **image**	John Napier, University of St Andrews
malcolm : **image**	Malcolm Atkinson, University of Glasgow
quintin : **image**	Quintin Cutts, University of St Andrews
richard : **image**	Richard Connor, University of St Andrews
ron : **image**	Ron Morrison, University of St Andrews
snoopy : **pic**	Snoopy the Beagle, Peanuts
stephan : **image**	Stephan Scheuerl, University of St Andrews
vivienne : **image**	Vivienne Moore, University of St Andrews

4.18 *Protection*

---

*protectedPackGen : **proc**[T](T → ProtectedPack[T])*

This procedure takes an initialising instance of the specialising type and returns a structure allowing instances of the specialising type to be protected. The components of the structure are:

---

protected : Protected

This is an abstract datatype whose witness type abstracts over the specialising type.

---

*setProtected : **any***

This encapsulates a procedure of type ***proc***(*Abs*) where *Abs* is the witness type of the abstract datatype *protected.* It records the given abstract reference to an instance of the specialising type.

---

getProtected : ***any***

This encapsulates a procedure of type ***proc(*** → *Abs)* where *Abs* is the witness type of the abstract datatype *protected*. It returns an abstract reference to the currently recorded instance of the specialising type.

---

setConcrete : ***proc(*** *T)*

This procedure records the given instance of the specialising type.

---

getConcrete : ***proc(*** → *T)*

This procedure returns the currently recorded instance of the specialising type in its concrete form.

The first program below illustrates how a value may be put into the persistent store in an abstract form:

```
let protectedIntPack = protectedPackGen[ int ]( 0 )

project PS() as root onto env :
begin
	in root let protectedIntPack = protectedIntPack

	use protectedIntPack( protected ) as X[ ProtectedInt ] in
	project protectedIntPack( getProtected ) as getProtectedInt onto
	proc( -> ProtectedInt ) :
	begin
		protectedIntPack( setConcrete )( 7 )
		in root let aProtectedInt = getProtectedInt()
	end
	default : writeString( "an error" )
end
default : {}
```

The next program illustrates how the value may be retrieved and converted back to its concrete form:

```
project PS() as root onto env :
use root with protectedIntPack : ProtectedPack[ int ] in
use protectedIntPack( protected ) as X[ ProtectedInt ] in
use root with aProtectedInt : ProtectedInt in
	project protectedIntPack( setProtected ) as setProtectedInt onto
	proc( ProtectedInt ) :
	begin
		setProtectedInt( aProtectedInt )
		let seven = protectedIntPack( getConcrete )()
	end
	default : writeString( "an error" )
default : {}
```

These procedures may be used to provide protected access to a set of values of a particular type as follows:

- call *protectedPackGen* specialised to the appropriate type;
- make the abstract datatype *protected* generally accessible; and

- restrict access to the other components of the structure, for example by password protection [CDM+90].

---

protectedBinding : Protected

This abstract datatype provides access to the protected type *Binding* which represents entities that may be passed to the browser or linked into hyper-programs. The code below shows an example of its use:

```
project PS() as root onto env :
use root with Library : env in
use Library with Browser,Protection : env in
use Protection with protectedBinding : Protected in
use protectedBinding as X[ Binding ] in
use Browser with graphicalBrowserGen : proc( WindowManager -> proc( Binding ) ) in
...
```

---

protectedTypeDescriptor : Protected

This abstract datatype provides access to the protected type *TypeDescriptor* which represents instances of type constructor information. It is used in the same way as *protectedBinding*.

---

protectedTypeRep : Protected

This abstract datatype provides access to the protected type *TypeRep* which represents types. It is used in the same way as *protectedBinding*.

4.19 *RasterRules*

This environment contains the following integers which represent raster rules for use with window operations:

copyRule	*andRule*	*orRule*	*xorRule*
notRule	*norRule*	*nandRule*	*xnorRule*

4.20 *String*

---

*length : **proc(string → int)***

This procedure returns the number of characters in the given string.

---

*asciiToString : **proc(int → string)***

This procedure returns the single character string corresponding to the ASCII code given by calculating *i* **rem** 128, where *i* is the parameter.

---

*stringToAscii : **proc(string → int)***

This procedure returns the ASCII code for the first character of the given string, unless the string is the empty string, in which case 0 is returned.

---

*letter : **proc(string → bool)***

This procedure returns **true** if the first character of the string is a lower case or upper case letter.

---

*digit : **proc(string → bool)***

This procedure returns **true** if the first character of the string is a decimal digit.

4.21 *System*

---

*stabilise : **proc()***

This procedure records the entire state of the Napier88 system on non-volatile storage. It is called automatically on normal program termination.

---

*diskgc : **proc()***

This procedure performs a garbage collection of the entire persistent store.

---

*abort : **proc()***

This procedure terminates the currently executing thread. No stabilisation is performed.

4.22 *Tables*

---

*compareInt : Comparison[**int**]*

This structure contains procedures to test equality and ordering on integers, and may be used with *tableGen* to generate tables keyed by integers.

---

*compareString : Comparison[**string**]*

This structure contains procedures to test equality and ordering on strings, and may be used with *tableGen* to generate tables keyed by strings.

---

*tableGen : **proc**[Key, Data](Comparison[Key] → Table[Key, Data])*

This procedure returns a structure containing procedures to manipulate an associative table with keys of type *Key* and associated data of type *Data*. The parameter is a variant that is either a structure containing procedures to compare key values for equality and ordering, or a structure containing only a procedure to test for equality. A more efficient implementation is obtained when both procedures are supplied. The procedures in the table structure are:

---

*enter : **proc**(Key, Data)*

This procedure inserts an entry into the table. If the table already contains an entry with the given key the existing value is overwritten.

---

*lookup : **proc**(Key → Optional[Data])*

This procedure returns either the data associated with the given key or the *absent* branch if the key is not found.

---

*remove : **proc**(Key)*

This procedure removes the data associated with the given key from the table. If the key is not found the procedure has no effect.

---

*scan : **proc**(**proc**(Key, Data → **bool**))*

This procedure calls the given procedure repeatedly, passing it each key present in the table and the associated data, until it has been called for all entries or it returns **false**. If the table has an ordering defined for it the entries are scanned in increasing key order. Otherwise the entries are scanned in the order in which the keys were inserted.

4.23 *Time*

---

*date : **proc**(→ **string**)*

This procedure gives the current date and time in the format illustrated below:

"Sat Oct 16 16:05:25 BST 1993"

---

*time : **proc**(→ **int**)*

This procedure returns the CPU time used by the Napier88 system since it was initialised. The time is measured in 60th of a second clock ticks.

4.24 *Utilities*

---

*concatenateHyperText : **proc**[T](HyperText[T], HyperText[T] → HyperText[T])*

This procedure concatenates the given fragments of hyper-text.

---

*concatenateStrings : **proc**(***string** → **string**)*

This procedure returns the string obtained by concatenating together the strings in the given vector.

---

*error : **proc(string)***

This procedure displays the given error message.

---

*executeAsThread : **proc(proc())***

This procedure executes the given procedure as a separate thread. The current thread is suspended until the new thread terminates. Any fatal run-time errors will not affect the current thread.

---

*extractHyperText : **proc**[T](HyperText[T], **int**, **int** → HyperText[T])*

This procedure extracts the part of the given fragment of hyper-text lying between the two given character positions inclusive.

---

*find : **proc(string**, **string**, **int** → **int**)*

This procedure searches the string given by the first parameter for the target string given by the second parameter, starting at the given offset into the string and wrapping back to the beginning if necessary. If the target is found the result is the offset at which the target occurs in the string. If the target is not found the result is zero.

---

*getArgs : **proc(→ *string**)*

This procedure returns the command line arguments used to invoke the current Napier88 session. The vector contains an element for each word, with a lower bound of 1. For example, if a session is initiated by typing

```
npr myProg.out arg1
```

at the command line, then a call to *getArgs* during the session will return the vector :

```
vector @1 of ["npr", "myProg.out", "arg1"]
```

---

*getEnv : **proc(→ *string**)*

This procedure returns the values of the shell environment variables in effect at the invocation of the current Napier88 session. The vector contains an element for each variable, with a lower bound of 1. An example is shown below:

```
vector @1 of ["DISPLAY=panda", "NPRDIR=/napier/release", "NPRSTORE=/napier/store"]
```

---

*getHyperProgramPack : **proc(bool**, **bool** → EditorPack[Binding])*

This procedure returns a structure containing procedures to operate on a hyper-program editor, as described for *hyperProgramPackGen* in Section 4.26.4. The first parameter specifies whether the contents of the editor can be edited interactively. The second parameter specifies whether the editor window contains *cut*, *copy* and *paste* buttons.

---

*getType : **proc(any** → TypeRep)*

This procedure returns an abstract representation of the type of the given value.

---

*max : **proc(int, int** → **int**)*

This procedure returns the maximum of the two integers.

---

*min : **proc(int, int** → **int**)*

This procedure returns the minimum of the two integers.

---

*mkBlankString : **proc(int** → **string**)*

This procedure returns a string containing the given number of space characters. If the parameter is negative the empty string is returned. The procedure is designed to minimise the number of object creations.

---

*mkCompareHyperText : **proc**[T](→ Comparison[HyperText[T]])*

This procedure returns a structure containing procedures to test equality and ordering on instances of hyper-text.

---

*mkEnvLocBinding : **proc(env, string** → Binding)*

This procedure returns a binding denoting the location with the given name in the given environment. If no such location exists then *bindingFailValue* is returned.

---

*mkHyperLink : **proc**[T](**string**, LinkPack[T] → HyperText[T])*

This procedure returns a fragment of hyper-text consisting of a single link to the given value with the given name.

---

*mkStructLocBinding : **proc(any, string** → Binding)*

This procedure returns a binding denoting the location with the given name in the given structure. If no such location exists then *bindingFailValue* is returned.

---

*mkTypeBinding : **proc**(TypeRep → Binding)*

This procedure converts the given type representation to a binding.

---

*mkTypeDescriptorBinding : **proc**(TypeDescriptor → Binding)*

This procedure converts the given type descriptor to a binding.

---

*mkValueBinding : **proc**(**any** → Binding)*

This procedure converts the given value to a binding.

---

*showBinding : **proc**(Binding, **int**)*

This procedure displays the binding denoted by the given abstract representation. The integer parameter is ignored.

---

*showType : **proc**(TypeRep → **string**)*

This procedure returns a string representation of the given abstract type representation.

---

*stringToHyperSource : **proc**(**string** → HyperText[Binding])*

This procedure returns a fragment of hyper-text consisting of the given string with no links.

---

*stringToInt : **proc**(**string** → **int**)*

This procedure converts the given string representation of an integer to the corresponding integer. If the string contains any non-digit characters other than a single leading "-" the result is zero.

4.25 *Vector*

---

*lwb : **proc**[t](*t → **int**)*

This procedure returns the lower bound of the vector.

---

*upb : **proc**[t](*t → **int**)*

This procedure returns the upper bound of the vector.

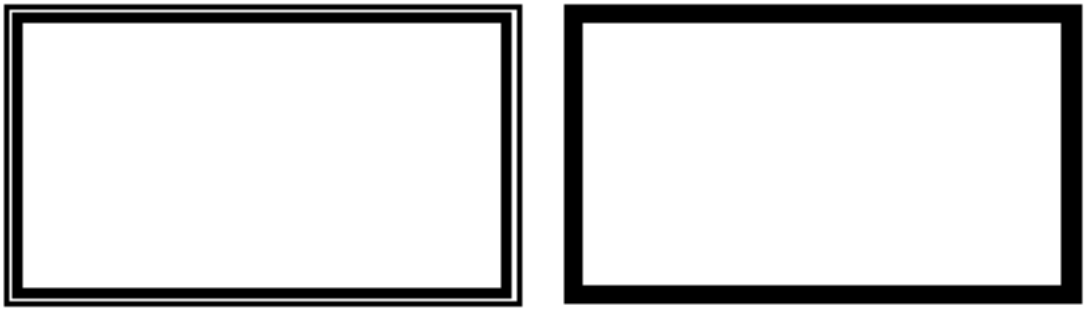
4.26 *Win*

4.26.1 *Borders*

---

double : BorderStyle

This border style produces a border with a double line around the window.



non-current current

Figure 4.2: *double* borders

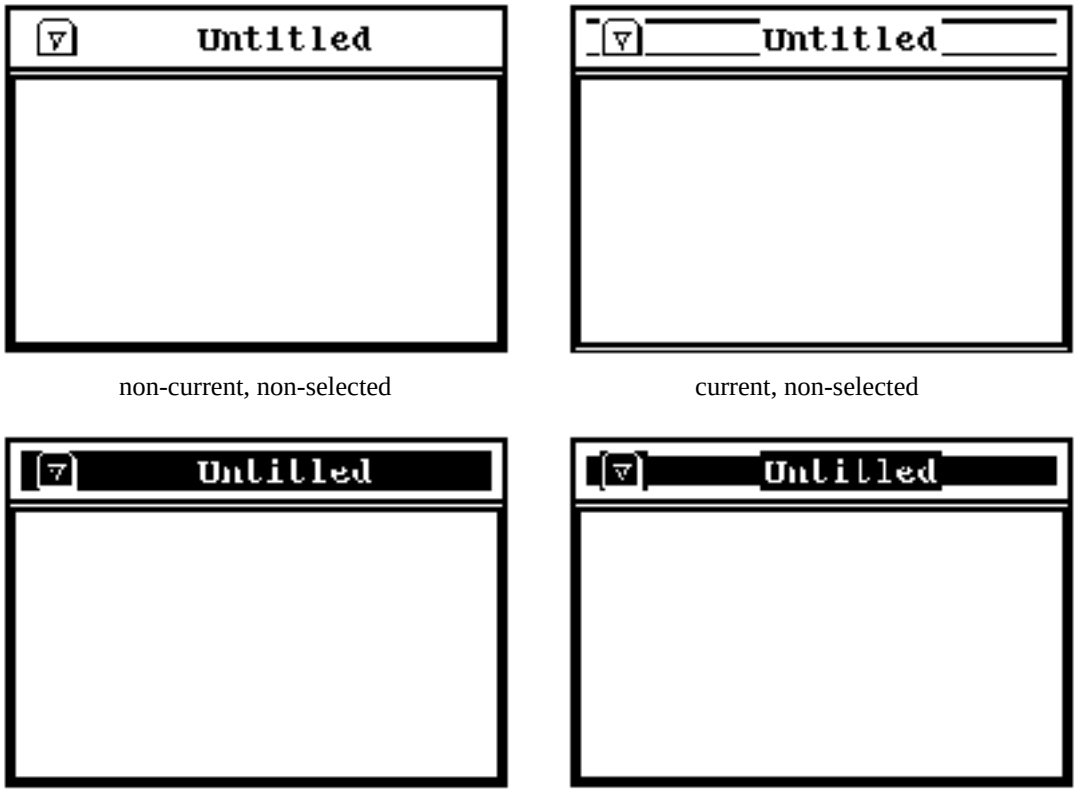
The interactive operations provided by the border are:

- The window can be brought to the front by clicking on the border with mouse button 1.
- The window can be moved by dragging the border with mouse button 2.
- The window can be undisplayed by clicking on the border with mouse button 3.

---

fixedX : BorderStyle

This border style produces an Open Look™ [Sun89] style border with a title bar and a close box.



non-current, non-selected current, non-selected

non-current, selected current, selected

Figure 4.3: *fixedX* borders

The interactive operations using the border are:

- The window can be selected or deselected by clicking on the border with mouse button 1. If the window is not already selected it becomes selected and any other selected windows are deselected. If the window is already selected it becomes deselected.

- The window can also be selected or deselected by clicking on the border with mouse button 2. In this case other selected windows are unaffected.

- A menu can be obtained by holding down mouse button 3 on the border. The entries in the menu are *Front*, to bring the window to the front, *Back*, to put the window to the back, and *Dismiss*, to undisplay the window.

- The window can be closed to its icon by clicking on the close box with mouse button 1.

This border style is equivalent to *genericXBorderGen(**false**, **true**, **true**, **true**)*.

---

*genericXBorderGen : **proc**(**bool**, **bool**, **bool**, **bool** → BorderStyle)*

This procedure produces a border style which in turn produces an Open Look style border. The first parameter specifies whether the border has a close box; the second parameter specifies whether the border has resize handles; the third parameter specifies whether a border menu is provided; the fourth parameter specifies whether a thin box is drawn around the inside of the border. Subject to these options the interactive operations on the border are the same as for *fixedX*.

---

invisible : BorderStyle

This border style produces a border with no visible parts and no interactive operations.

---

menuX : BorderStyle

This border style produces the same border as that produced by *fixedX*, without a close box or inner rectangle.

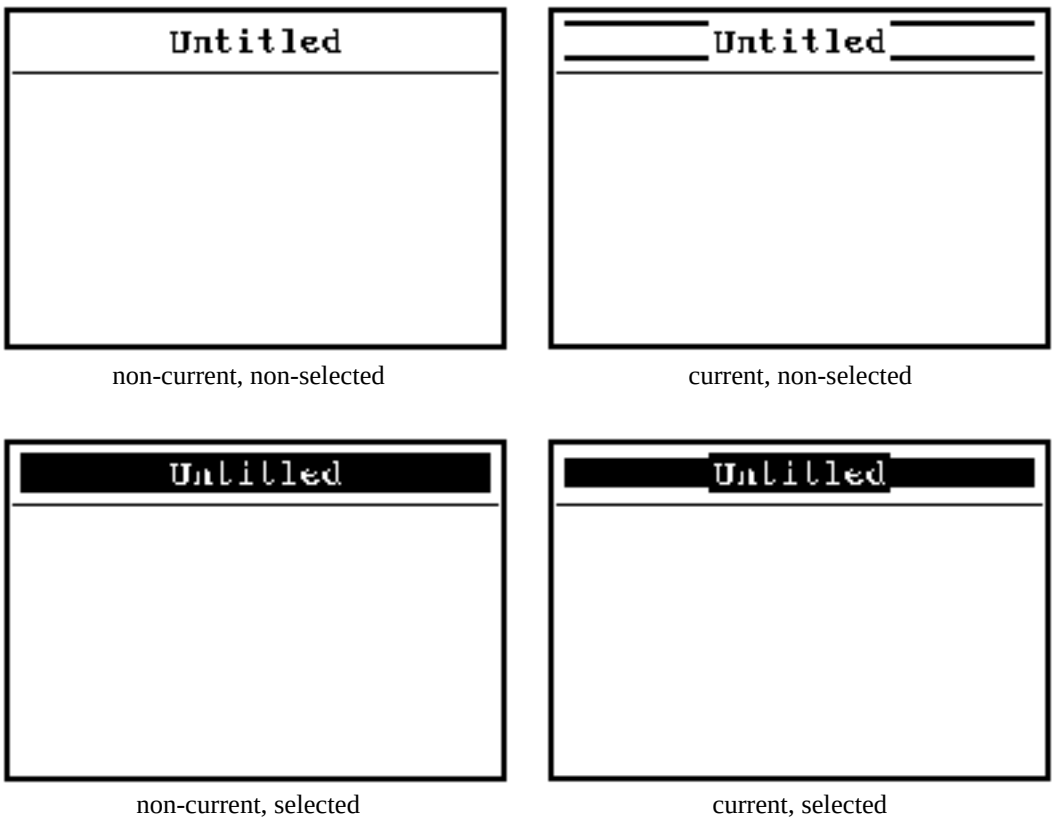


Figure 4.4: *menuX* borders

This border style is equivalent to *genericXBorderGen(**false**, **false**, **false**, **false**)*.

---

plain : BorderStyle

This border style produces a border with a single line around the window and no interactive operations.

Figure 4.5: A *plain* border

---

shadow : BorderStyle

This border style produces a border with a single line and a shadow around the window and no interactive operations.

Figure 4.6: A *shadow* border

---

variableX : BorderStyle

This border style produces the same border as that produced by *fixedX*, with the addition of resize handles at the four corners.

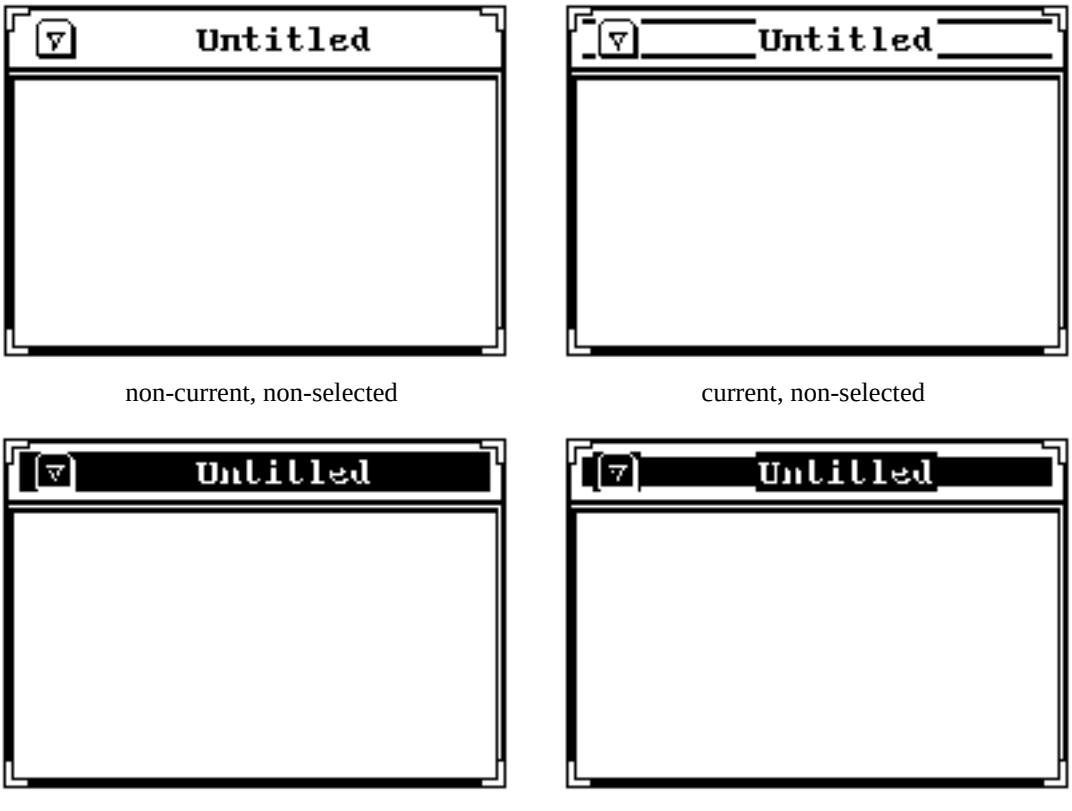


Figure 4.7: *variableX* borders

The window can be resized by dragging a resize handle with mouse button 1.

This border style is equivalent to *genericXBorderGen(**true**, **true**, **true**, **true**)*.

4.26.1.1 *Interactive*

This section describes how the user can define new styles of borders. A border is defined by splitting it up into a number of areas using the following types:

```
type BorderStyle is proc( Window -> Border )
```

```
type Border is      List[ Area ]
type Area is        structure(  currentImage,nonCurrentImage : image ;
                                pos : Pos ; distributeEvent : Application )
```

A border style is represented as a procedure which takes as its parameter a window and returns a list of values of type *Area*. Each of these is a structure that contains two images for a part of the border. One of the images is displayed when the window is current and the other when it is not. The border as a whole is built up from the separate areas. Each structure also contains the position of the origin of the area relative to the origin of the window and an application that processes mouse events that occur over the area. No keyboard events are sent to border applications. Mouse events sent to border applications are translated so that the positions are relative to the origin of the window.

A border style could be split into four areas as illustrated in Figure 4.8:

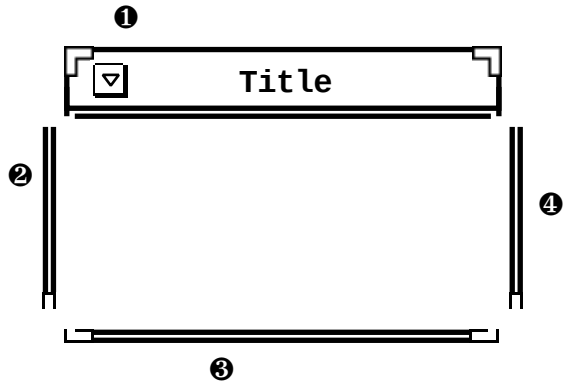


Figure 4.8: Border style areas

A new border style is made by constructing a procedure of type *BorderStyle* which takes a window as its parameter and returns a list of border areas. The procedure will use the size of the window to calculate the sizes and positions of the areas. The only restriction on the appearance of a border style is that its outline should be rectangular. If this is not adhered to the border may not be displayed correctly.

A number of predefined procedures are available for performing interactive window manipulation and these can be incorporated into a new border style. The *move* procedure, for example, displays an outline of the window and moves it around following the position of the mouse until the mouse button is released, when it calls the window manager's move procedure to move the window to its new position.

The procedures can be incorporated into a border style by using them within the applications for the border areas.

---

delete : ***proc****(Window, Event, EventTest,* ***proc****(),* ***proc****())*

This procedure undisplays the given window. The second parameter is ignored and is present only for compatibility with the other procedures in the environment. Before undisplaying the window, *delete* calls the first of the void procedure parameters. After undisplaying the window it loops until the *EventTest* parameter returns **false** on the current event and then calls the second of the void procedure parameters.

A typical use of this procedure is for a border style to call it when a mouse down event is to be interpreted as a 'delete window' command.

---

iconise : ***proc****(Window, Event, EventTest,* ***proc****(),* ***proc****())*

This procedure closes the given window. The second parameter is ignored and is present only for compatibility with the other procedures in the environment. Before closing the window, *iconise* calls the first of the void procedure parameters. After closing the window it loops until the *EventTest* parameter returns **false** on the current event and then calls the second of the void procedure parameters.

A typical use of this procedure is for a border style to call it when a mouse down event is to be interpreted as a 'close window' command.

---

move : ***proc****(Window, Event, EventTest,* ***proc****(),* ***proc****())*

This procedure allows the given window to be moved interactively. The *Event* parameter is interpreted as the event which caused the procedure to be invoked. Before moving the window, *move* calls the first of the void procedure parameters. It then displays an outline of the given window and allows it to be dragged by the mouse until the *EventTest* parameter returns **false** on the current event. It then moves the window to the current position of the outline and calls the second of the void procedure parameters.

A typical use of this procedure is for a border style to call it when a mouse down event is to be interpreted as a 'move window' command.

---

pushPop : ***proc****(Window, Event, EventTest,* ***proc****(),* ***proc****())*

This procedure brings the given window to the front, unless the window is already at the front, in which case it sends the window to the back. The second parameter is ignored and is present only for compatibility with the other procedures in the environment. Before moving the window, *pushPop* calls the first of the void procedure parameters. After moving the window it loops until the *EventTest* parameter returns **false** on the current event and then calls the second of the void procedure parameters.

A typical use of this procedure is for a border style to call it when a mouse down event is to be interpreted as an 'alter window depth' command.

---

resize : ***proc****(Window, Event, EventTest,* ***proc****(),* ***proc****())*

This procedure allows the given window to be resized interactively. The second parameter is ignored and is present only for compatibility with the other procedures in the environment. Before resizing the window, *resize* calls the first of the void procedure parameters. It then displays an outline of the given window and allows its size to be altered by dragging the mouse until the *EventTest* parameter returns **false** on the current event. If the cursor is initially close to a corner of the window then the position of that corner is altered during the resize. Otherwise the position of the window edge nearest the cursor is altered. The procedure then moves the window to the current position of the outline and calls the second of the void procedure parameters.

---

windowCreatorGen : ***proc****(WindowManager,* ***proc****(Size, WindowManager,* ***int*** *→ Window), Size → Application)*

This procedure returns an application which allows a rectangular outline to be dragged out using any mouse button, subject to the minimum size given by the third parameter. When the

mouse button is released the application creates a window by calling the second parameter and displays it at the position of the outline. The window manager parameter should be the window manager running in the window in which the application is set. The integer parameter passed to the second parameter is the number of the mouse button being used.

4.26.2 *CurrentState*

---

currentBuffer : Editor[Binding]

This variable contains a cut/copy/paste buffer shared among hyper-program editors created by calling *hyperProgramPackGen* described in Section 4.26.4.

---

currentError : ***proc****(* ***string*** *)*

This variable contains a procedure which may be used to display error messages.

---

currentOutputPack : EditorPack[Binding]

This variable contains an editor which may be used to display hyper-program messages.

---

currentWindowManager : WindowManager

This variable contains the top-level window manager currently active.

---

currentWriteString : ***proc****(* ***string*** *)*

This variable contains a procedure which may be used to display string messages.

4.26.2.1 *CurrentBrowser*

---

browser : ***proc****(Binding)*

This procedure displays the given value using the currently active browser.

---

deselect : ***proc****(Binding)*

This procedure deselects the given value. It is removed from the selection list and its browser representation assumes the unselected appearance.

---

select : ***proc****(BindingInfo[Binding],* ***bool*** *)*

This procedure selects the value in the given structure. If the boolean parameter is **false** any other selected values are deselected. The selected value is added to the selection list and its browser representation assumes the selected appearance. The other structure fields contain a name for the binding, which may be blank, and, for a location, the vertical offset from the base of the parent menu both in pixels and in numbers of entries. For non-locations these fields are ignored.

---

*selected : **proc**(Binding → **bool**)*

This procedure returns **true** iff the given value is in the selection list.

---

*getSelectedBindingInfo : **proc**(→ List[BindingInfo[Binding]])*

This procedure returns the current selection list.

4.26.3 *Defaults*

---

defaultBoldFont : FontPack

This variable contains the default bold font. Initially it is set to *courB14*.

---

defaultBorderStyle : BorderStyle

This variable contains the default border style. Initially it is set to *variableX*.

---

*defaultBorderThickness : **int***

This variable contains the default border thickness. Initially it is set to 2.

---

*defaultColourMap : ***int***

This variable contains the default colour map used when the programming environment is initialised.

---

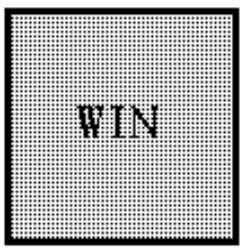
defaultFont : FontPack

This variable contains the default font. Initially it is set to *courR14*.

---

*defaultIcon : **image***

This variable contains the default icon image. Initially it is set to the image shown below:



---

*defaultLinkMarker : **string***

This variable contains the marker used by hyper-program editors to denote embedded link information in files. Initially it is set to "!@£$". See Section 4.26.4.

*defaultOuterScopes : ****string***

This variable contains the names of the default declaration sets associated with a newly created hyper-program editor. Initially it is set to *stringVectorFailValue*, denoting no declaration sets.

---

defaultWindowCursor : ***image***

This variable contains the default cursor image. Initially it is set to the image shown below:

---

defaultWindowDepth : ***int***

This variable contains the default image depth for windows. Initially it is set to 1.

---

defaultWindowSize : Size

This variable contains the default size for windows. Initially it is set to *Size(300, 200)*.

---

defaultXWindowPos : Pos

This variable contains the default position for X windows relative to the top left corner of the screen. Initially it is set to *Pos(10, 10)*.

---

defaultXWindowSize : Size

This variable contains the default size for X windows. Initially it is set to *Size(1100, 770)*.

4.26.4 *Generators*

---

editorGen : ***proc****[HyperLink](Window → Editor[HyperLink])*

This procedure returns a hyper-text editor running in the given window. The type parameter specifies the type of the hyper-links which may be embedded in the text of the editor. The editor is implemented as an abstract data type containing procedures which fall into several categories:

- procedures for reading and writing hyper-text to and from the editor;
- procedures for navigating around the hyper-text in the editor;
- procedures for controlling interactive input and the appearance of the window display; and
- procedures for setting and reading attributes of the editor.

Although the editor returned by this procedure operates on a window it is possible to de-couple the editor from its window and later re-couple it to the same or a different window.

This makes it possible to retain text in the persistent store within an editor without the potentially large overhead of storing the associated window by storing the editor in its unbound state. While an editor is de-coupled from its window the procedures specific to the window display are disabled. Operations on the hyper-text may still be performed but the effects are not visible in any window.

An editor manipulates a number of lines of hyper-text separated by carriage returns. There is no limit on the length of a line. Each line may contain both characters and links to instances of the parameter type *HyperLink*. The editor records the **current selection**, which is a pair of points in the hyper-text, all the hyper-text between the points being considered as selected. The points may coincide, in which case the selection is empty. Editing functions such as *cut*, *copy* and *paste* operate on the current selection.

Instances of the witness type of the abstract type *Editor* are used to represent positions within the hyper-text. The user is unable to perform any operations on such values except to use them as parameters to editor functions. Thus the user cannot discover the internal representation of the hyper-text or manipulate it other than through the editor interface.

If a line of text is too long to fit on one line in the window, it wraps onto subsequent window lines. No word wrapping is performed. The text is displayed in a single fixed-width font only. The font can be set by the user. The user can turn highlighting on or off. When it is on, the text currently selected (if visible in the window) is shown in inverse video. When it is off the selection is displayed as normal, although it is still possible for the user to invert arbitrary regions of text using the *invert* procedure.

The editor's initial font is given by the current value of *defaultFont* in the environment *Defaults*.

The procedures in an editor with witness type *TextPointer* and parameterised by type *HyperLink* are:

---

copyText : ***proc****(Editor[HyperLink])*

This procedure takes as parameter another editor of the same type to act as a buffer, and copies the currently selected text to the buffer. The current selection is unaffected but the previous contents of the buffer are over-written.

---

cutText : ***proc****(Editor[HyperLink])*

This procedure performs the same actions as *copyText* except that the current selection is deleted.

---

clearText : ***proc****()*

This procedure deletes the current selection.

---

pasteText : ***proc****(Editor[HyperLink])*

This procedure over-writes the current selection with the contents of the given buffer.

---

*insertText : **proc**(HyperText[HyperLink], **bool**)*

This procedure over-writes the current selection with the given hyper-text. If the boolean parameter is **true** the window display is updated incrementally as the text is inserted. If the parameter is **false** the display is updated only after the insertion has been completed. This option gives better performance when large sections of hyper-text are inserted. If no window is currently coupled to the editor, the boolean parameter has no effect.

---

*readFromFile : **proc**(**file**)*

This procedure over-writes the current selection with the contents of the given file. This will result in text only without any hyper-links.

---

*writeToFile : **proc**(**file**)*

This procedure writes the textual contents of the editor out to the given file. The title of each hyper-link button is written out at the appropriate position.

---

*select : **proc**(TextPointer, TextPointer)*

This procedure sets the current selection to the hyper-text between the given positions. Note that text positions cannot be created by the user but can only be obtained by calling editor procedures.

---

*firstSelection : **proc**(→ TextPointer)*

This procedure returns the starting position of the current selection.

---

*lastSelection : **proc**(→ TextPointer)*

This procedure returns the finishing position of the current selection.

---

*firstLine : **proc**(→ TextPointer)*

This procedure returns the position of the beginning of the first text line.

---

*lastLine : **proc**(→ TextPointer)*

This procedure returns the position of the beginning of the last text line.

---

*topLine : **proc**(→ TextPointer)*

This procedure returns the position of the start of the top-most window line. This need not be at the beginning of a text line. If no window is currently coupled to the editor an error is reported using the procedure *error* as described in Section 4.26.8 and the position returned is the beginning of the first text line.

---

bottomLine : ***proc****(→ TextPointer)*

This procedure returns the position of the start of the bottom-most window line. This need not be at the beginning of a text line. If no window is currently coupled to the editor an error is reported using the procedure *error* as described in Section 4.26.8 and the position returned is the beginning of the first text line.

---

frontOfLine : ***proc****(TextPointer → TextPointer)*

This procedure returns the position of the beginning of the text line containing the given position.

---

endOfLine : ***proc****(TextPointer → TextPointer)*

This procedure returns the position of the end of the text line containing the given position.

---

nextLine : ***proc****(TextPointer → TextPointer)*

This procedure returns the position of the beginning of the text line following the text line containing the given position. If the given position is in the last text line, the position of the end of that line is returned.

---

previousLine : ***proc****(TextPointer → TextPointer)*

This procedure returns the position of the beginning of the text line preceding the text line containing the given position. If the given position is in the first text line, the position of the beginning of that line is returned.

---

peek : ***proc****(→ HyperText[HyperLink])*

This procedure returns the next character or hyper-link after the current selection, or the empty string if the current selection is at the end of the hyper-text. A newline character is returned if the current selection ends at the end of a text line.

---

read : ***proc****(→ HyperText[HyperLink])*

This procedure performs the same actions as *peek* except that the current selection is advanced so that it begins and ends at the point after the character or hyper-link read, unless it is already at the end of the hyper-text.

---

readLine : ***proc****(→ HyperText[HyperLink])*

This procedure returns the remainder of the text line after the current selection, not including the newline character at the end, and advances the current selection to the beginning of the following text line. If it already ends in the last line, the current selection is moved to the end of that line.

---

selectedText : ***proc****(→ HyperText[HyperLink])*

This procedure returns the hyper-text in the current selection.

---

before : ***proc****(TextPointer, TextPointer →* ***bool*** *)*

This procedure returns **true** iff the first position lies before the second position in the hyper-text.

---

endOfText : ***proc****(→* ***bool*** *)*

This procedure returns **true** iff the end of the current selection is at the end of the last text line.

---

getFont : ***proc****(→ FontPack)*

This procedure returns the font used to display the hyper-text.

---

getHighlight : ***proc****(→* ***bool*** *)*

This procedure returns **true** iff the current selection is highlighted.

---

getProgressIndicator : ***proc****(→* ***proc****(* ***string****,* ***real*** *))*

This procedure returns the user-set procedure used to display file I/O progress.

---

getScrollAction : ***proc****(→* ***proc****(* ***int****,* ***bool*** *))*

This procedure returns the user-set procedure that is called whenever the hyper-text is scrolled.

---

getWindow : ***proc****(→ Window)*

This procedure returns the window currently coupled to the editor or a fail value if there is no such window.

---

interactiveEdit : ***proc****(Editor[HyperLink], EventTest, EventTest, EventTest → Application)*

This procedure generates an application which can be used to allow the user to enter and edit text in the window interactively. It takes as parameters an editor to use as an editing buffer and procedures to detect *point*, *extend* and other mouse button presses. Note that the application for the window must be set explicitly to allow interactive editing.

The interactive operations supported by the application are as follows:

operation	method
enter text	type at keyboard
position insertion point	click mouse button 1
set current selection	drag region of text with mouse button 2
extend current selection	click with mouse button 2
select word	double click with mouse button 1
delete current selection	type ‘backspace’ or ‘delete’
cut current selection	type ‘ctrl-x’
copy current selection	type ‘ctrl-c’
paste	type ‘ctrl-v’

Table 4.3: Interactive operations

On some terminals the codes for cut, copy and paste may not work due to the characters being trapped before reaching the WIN system.

---

invert : ***proc****(TextPointer, TextPointer)*

This procedure inverts the pixels of the characters between the two given text positions.

---

lineCount : ***proc****(→ **int**)*

This procedure returns the number of text lines currently held in the editor.

---

new : ***proc****()*

This procedure deletes all the text in the editor.

---

offset : ***proc****(TextPointer,* ***bool*** *→ Index)*

This procedure returns the distance from the start of the text to the given position. The distance is returned as a variant, being the number of characters if the given boolean value is **true** and otherwise a structure containing the number of lines and an offset within a line.

---

position : ***proc****(Pos → TextPointer)*

This procedure returns the position in the text which is currently closest to the given position in the window display relative to the bottom left of the window.

redisplay : ***proc****(TextPointer)*

This procedure redraws the contents of the window with the given text position at the top. If the text position is not at the beginning of a text line, the procedure uses instead the first text position before the given one which would normally fall at the beginning of a window line. This ensures that a text line always starts at the beginning of a window line.

---

scanLinks : ***proc****(* ***proc****(Substitution[LinkPack[HyperLink]] →* ***bool*** *))*

This procedure applies the given procedure successively to each link in the hyper-text, in their order within the hyper-text, until either the procedure returns **false** or it has been called for all the links.

---

scroll : ***proc****(* ***int****,* ***bool*** *)*

This procedure scrolls the window display up or down. If the boolean parameter is **true** then the display is scrolled up by the given number of window lines, a negative number giving downwards scrolling. If the boolean is **false** then the integer parameter gives the absolute position to scroll to, as a number of text lines from the beginning of the text.

---

search : ***proc****(HyperText[HyperLink],* ***bool*** *→* ***bool*** *)*

This procedure searches for the given hyper-text, starting from the end of the current selection, moving forwards or backwards depending on the given boolean value. The boolean value returned is **true** iff the hyper-text is found, in which case the hyper-text is selected. A link in the target hyper-text matches a link in the editor iff they are identical.

---

seek : ***proc****(Index → TextPointer)*

This procedure returns the position corresponding to the given offset from the start of the text.

---

setFont : ***proc****(FontPack)*

This procedure sets the font used to display the text. The font is checked to make sure that it contains all the necessary characters and that they are all the same size. If so, the contents of the window are redrawn in the new font, otherwise the procedure has no effect.

---

setHighlight : ***proc****(* ***bool*** *)*

This procedure sets the highlight on if the argument is **true**, or off otherwise. When the highlight is on the current selection is inverted whenever it is visible in the window.

*setProgressIndicator : **proc(proc(string, real))***

This procedure sets the procedure used to display file I/O progress. The string parameter specifies the type of I/O and the real parameter specifies the fraction of the I/O completed.

---

*setScrollAction : **proc(proc(int, bool))***

This procedure sets the procedure that is called whenever the hyper-text is scrolled. The parameters are the same as those passed to *scroll*.

---

*setWindow : **proc(Window)***

This procedure sets the display window for the editor. Any existing contents of the window are erased and the text is displayed in it starting from the beginning of the first text line. If the editor is already coupled to a window the procedure has no effect.

---

*unbindWindow : **proc()***

This procedure de-couples the editor from its display window. If there is no window coupled to the editor the procedure has no effect.

---

*eventMonitorGen : **proc(proc(→ bool), Application → proc())***

This procedure returns a procedure (an event monitor) that repeatedly gathers user input events and passes them to the given application. It calls the first procedure parameter after passing each event, and terminates when that procedure returns **true**.

This could be used, for example, to initiate a WIN session without starting the programming environment, as illustrated below:

```
let aWindowManager = screenWindowManagerGen( 1 )
in root let finished := false       ! make variable accessible by other programs
let em = eventMonitorGen( proc( -> bool ) ; finished,
                          aWindowManager( getDisplayWindow )()( getApplication )() )
em()
```

---

*hyperProgramPackGen : **proc(** Size, **bool**, **bool**, **proc(** → List[Table[**string**, Binding]])*
→ EditorPack[Binding])

This procedure returns an instance of *EditorPack* specialised to links of type *Binding*. The first parameter specifies the size of the window. The contents of the editor are interactively editable iff the second parameter is **true**. Iff the third parameter is **true** the window contains all the light-buttons described in Section 2.1.

The fourth parameter is a procedure that returns a list of tables mapping string names to *Binding*s. This list is used to form a series of outer scopes during compilation of the editor contents.

When the contents of the hyper-program editor are written out to a file the editor records, where possible, information about the positions in the store of the *Binding* links. In some

cases this enables the links to be reconstructed when the hyper-program is read back from the file. Such a case arises when a *Binding* is accessible through a chain of environments from the root of persistence. Since, however, the information recorded for each such case is the path from the persistent root, there is no guarantee that the reconstructed link is the same as the original.

The string *defaultLinkMarker*, described in Section 4.26.3, is used to indicate the presence of a link record in the file. It may be updated if it clashes with genuine text in the editor. If it is set to the empty string then the editor does not attempt to record or interpret any link information on file writes and reads.

The resulting *EditorPack* contains the following fields:

---

window : Window

This window contains the editor window and associated scroll bar and light-buttons.

---

editor : Editor[Binding]

This is the editor itself.

---

getTitle : ***proc****(* → ***string*** *)*

This procedure returns the title of the source code currently being edited.

---

getText : ***proc****(* → *HyperText[Binding])*

This procedure returns the entire contents of the editor.

---

append : ***proc****(HyperText[Binding])*

This procedure inserts the given hyper-text after the existing contents of the editor.

---

screenWindowManagerGen : ***proc****(* ***int*** → *WindowManager)*

This procedure returns a window manager which operates directly on the screen or on an X window. The parameter gives the depth of the display area in planes. The procedure first attempts to open the screen as a raw device. If this fails, for example because a (non-Napier) window manager is running, the procedure attempts to connect to the X-server indicated by the UNIX environment variable DISPLAY and to create an X window on which to operate the window manager. If this also fails a fail value is returned.

---

unboundEditorGen : ***proc****[HyperLink](* → *Editor[HyperLink])*

This procedure returns an editor with no display window coupled to it.

*windowGen : **proc**(→ Window)*

This procedure returns a window. The window's initial icon, cursor, size, pixel depth and border style are given respectively by the current values of *defaultIcon*, *defaultWindowCursor*, *defaultWindowSize*, *defaultWindowDepth* and *defaultBorderStyle* in the environment *Defaults*.

The fields of the window are as follows:

---

*windowRaster : **proc**(Limit, Limit, Window, **int**, **bool**)*

This procedure performs a raster operation between the window and another given window. The first limit specifies the region in the window and the second the region in the other window. The integer parameter specifies the raster rule to be used according to the values in the *RasterRules* environment.

The boolean parameter specifies the direction of the raster operation. If it is **true** the raster operation is from the other window to the window, otherwise the operation is from the window to the other window. If the source region is larger than the destination region it is clipped on the top and right sides as necessary. If it is smaller than the destination the new pixels are drawn starting at the bottom left of the destination region.

For example, the following code **xor**s an area of 10 by 10 pixels starting at position (0,0) from *window1* onto *window2* at the position (10,20):

```
let destination = Limit( Pos( 10,20 ), Size( 10,10 ) )
let source = Limit( Pos( 0,0 ),Size( 10,10 ) )
window2( windowRaster )( destination, source, window1, xorRule, true )
```

---

*imageRaster : **proc**(Limit, **image**, **int**, **bool**)*

This procedure performs a raster operation between the window and a given image. The limit parameter specifies the region in the window. The integer parameter specifies the raster rule to be used according to the values in the *RasterRules* environment.

The boolean parameter specifies the direction of the raster operation. If it is **true** the raster operation is from the image to the window, otherwise it is from the window to the image. Clipping of the window region is performed as for *windowRaster*.

---

*drawLine : **proc**(Pos, Pos, **pixel**, **int**)*

This procedure draws a line on the window between the given points, using the given pixel and raster rule. If either point lies outside the window the line is clipped to the boundaries of the window.

---

*setInputOption : **proc**(InputOption)*

This procedure specifies how the window receives input events when it is the current window displayed by a window manager. The parameter is interpreted according to its branch as follows:

all : the window receives all input events detected by the window manager until the input option is reset;

normal : the window receives mouse events over the window region and all text events;

none : the window receives no input events until the input option is reset.

---

*getInputOption : **proc**(→ InputOption)*

This procedure returns the input option currently associated with the window.

---

*setSize : **proc**(Rect)*

This procedure changes the size of the window to that of the specified rectangle. The rectangle's coordinates are given relative to the current origin of the window. The existing contents of the window are redrawn at the old origin of the window, after being clipped if necessary. The bottom left corner of the resized window becomes the new origin of the window's coordinate system.

For example, the following code creates a window of the default size (assumed to be 100 by 100 pixels) and then enlarges it by 10 pixels in both directions. Blank space is added at the left and bottom of the window and the old contents are drawn on the window starting at the point (10,10).

```
let windowOne = windowGen()
windowOne( setSize )( Rect( Pos( -10, -10 ), Pos( 100, 100 ) ) )
```

---

*getSize : **proc**(→ Size)*

This procedure returns the current size of the window.

---

*setApplication : **proc**(Application)*

This procedure sets the application for the window. The application is a procedure which takes an input event as its parameter and performs some action.

---

*getApplication : **proc**(→ Application)*

This procedure returns the application currently associated with the window.

---

*setTitle : **proc**(**string**)*

This procedure sets the title for the window.

---

*getTitle : **proc**(→ **string**)*

This procedure returns the title currently associated with the window.

*setResizeControl : **proc**(ResizeControl)*

This procedure sets the resize behaviour for the window. It allows the programmer of an application to specify how to regenerate the display when the window in which it is running is resized. The fields of the structure parameter are interpreted as follows:

> *before : **proc**(Rect → Rect)*
>
> This procedure is called immediately the window's *setSize* procedure is called, before any changes are made to the window. The parameter gives the proposed new size of the window. The procedure may perform any actions necessary before the window is resized. The result of the procedure is the actual permitted new size of the window, which may be different from the proposed new size. If the result is equal to *rectFailValue* in the environment *FailValues* the entire resize operation is vetoed.
>
> *after : **proc**(Rect)*
>
> This procedure is called after a window has been resized, with the new size passed to it.

For example, the following code shows the procedures being set for a window whose application displays a view onto an image which is larger than the window. To conserve memory the application does not keep a separate copy of the part of the image which is shown in the window. The procedure *before* copies from the window any parts of the image that will cease to be visible, while *after* draws on any parts of the image that have newly become visible.

```
let resizeControl =
begin
        let before = proc( newRect : Rect → Rect )
        begin
                …     !*** Copy parts that become invisible after resize.
                newRect
        end

        let after = proc( newRect : Rect )
        begin
                …     !*** Fill in parts that become visible after resize.
        end

        ResizeControl( before,after )
end

imageViewerWindow( setResizeControl )( resizeControl )
```

*getResizeControl : **proc**(→ ResizeControl)*

This procedure returns the resize control structure currently associated with the window.

*setMinSize : **proc**(Size)*

This procedure sets the minimum size to which the window can be resized. Subsequent calls to *resize* with a size smaller than the minimum in either direction will have no effect. If the window is currently smaller than the minimum specified then the minimum is not set.

---

*getMinSize : **proc**(→ Size)*

This procedure returns the minimum size currently associated with the window.

---

*setMaxSize : **proc**(Size)*

This procedure sets the maximum size to which the window can be resized. Subsequent calls to resize with a size larger than the maximum in either direction will have no effect. If the window is currently larger than the maximum specified then the maximum is not set.

---

*getMaxSize : **proc**(→ Size)*

This procedure returns the maximum size currently associated with the window.

---

*setDepth : **proc**(**int**)*

This procedure sets the pixel depth of the window. If the new depth is greater than the existing depth then planes of **off** are added behind the existing planes. If the new depth is less than the existing depth then planes from the back are discarded.

---

*getDepth : **proc**(→ **int**)*

This procedure returns the current pixel depth associated with the window.

---

*setBorderStyle : **proc**(BorderStyle)*

This procedure sets the border style used to display the window in a window manager.

---

*getBorderStyle : **proc**(→ BorderStyle)*

This procedure returns the current border style associated with the window.

---

*getBorder : **proc**(→ Border)*

This procedure returns the border currently associated with the window. If the window is not displayed by a window manager the list is empty.

*setCursor : **proc(image)***

This procedure sets the image displayed when the cursor moves over the window.

---

*getCursor : **proc(→ image)***

This procedure returns the cursor image currently associated with the window.

---

*getWindowManager : **proc(→ WindowManager)***

This procedure returns the window manager currently displaying the window. If the window is not displayed the fail value *windowManagerFailValue* is returned.

---

*setVirtualWindow : **proc(string**, Window)*

This procedure is for system use only and is password protected.

---

*windowManagerGen : **proc(Window → WindowManager)***

This procedure returns a window manager operating in the given parent window. The fields of the window manager are as follows:

---

*display : **proc(DisplayInfo, bool)***

This procedure displays a window. The first parameter is a structure containing the window, its required position on the window manager display relative to the bottom left corner and its level relative to other windows. Iff the boolean parameter is **true** the window is displayed in the background behind all other windows. The window is not displayed if it is already displayed by another window manager or if an attempt is made to display it in the background when a background window already exists.

---

*undisplay : **proc(Window)***

This procedure removes the given window from the window manager display.

---

*makeCurrent : **proc(Window)***

This procedure sets the window manager's current window to be the specified window. Any existing current window is made non-current.

---

*setPos : **proc(Window, Pos)***

This procedure moves the given window to the given position.

---

*getPos : **proc**(Window → Pos)*

This procedure returns the position of the origin of the given window relative to the bottom left corner of the parent window.

---

*setLevel : **proc**(Window, Level)*

This procedure moves the given window to the given level relative to other windows. If the boolean field in the second parameter is **true** then the integer parameter is interpreted as the number of windows from the front, otherwise as the number of windows from the back. Thus Level(**true**, 1) puts the window at the front, while Level(**false**, 2) puts it second from the back.

---

*getLevel : **proc**(Window, **bool** → Level)*

This procedure returns the level of the given window. If the boolean parameter is **true** the result is relative to the front, otherwise it is relative to the back. Any background window is ignored.

---

*getWindows : **proc**(→ *Window)*

This procedure returns a vector containing the windows currently displayed by the window manager, with a lower bound of 1, starting with the window nearest the front.

---

*getWindowAtPos : **proc**(Pos → Window)*

This procedure returns the front-most window which overlaps the given position. If there is none then *windowFailValue* is returned.

---

*getNotifier : **proc**(string → Notifier)*

This procedure is for system use only and is password protected.

---

*getDisplayWindow : **proc**(→ Window)*

This procedure returns the window in which the window manager is running.

---

*getIconManager : **proc**(→ IconManager)*

This procedure returns an icon manager for the window manager. The fields are as follows:

> *close : **proc**(Window)*
>
> This procedure undisplays the given window and displays an icon in its place, itself a window.

*open : **proc**(Window)*

This procedure undisplays the given icon and redisplays the corresponding window at its original position.

*getIconState : **proc**(Window → DisplayInfo)*

This procedure returns a structure containing information about the icon associated with the given window. The fields of the structure can be updated to change the way that the icon will be displayed when the window is next closed.

*getWindowState : **proc**(Window → DisplayInfo)*

This procedure returns a structure containing information about the window associated with the given icon. The fields of the structure can be updated to change the way that the window will be displayed when the icon is next opened.

---

*setBackgroundApp : **proc**(Application)*

This procedure sets an application to run in the background of the window manager. The application receives keyboard events when there is no current window, and mouse events which do not occur over a window.

---

*getBackgroundApp : **proc**(→ Application))*

This procedure returns the current background application.

4.26.5 *Images*

This environment contains the following images:

---

borderCursor : ***image*** 

downArrow : ***image*** 

iconiseCursor : ***image*** 

leftArrow : ***image*** 

magnify : ***image*** 

moveCursor : ***image*** 

resizeCursor : ***image*** 

rightArrow : ***image*** 

upArrow : ***image*** 

waitCursor : ***image*** 

windowIcon : ***image*** 

4.26.6 *Selection*

---

deselect : ***proc****(Window)*

This procedure deselects the given window. If the window is not already selected the procedure has no effect.

---

select : ***proc****(Window,* ***bool*** *)*

This procedure selects the given window. If the window is already selected the procedure has no effect. If the boolean parameter is **true** the procedure does not affect other windows. If the parameter is **false** any other selected windows are deselected.

---

selected : ***proc****(Window →* ***bool*** *)*

This procedure returns **true** iff the given window is selected.

---

selectedWindows : List[Window]

This list contains the windows that are currently selected.

*refinements : Table[WindowManager, **proc**(Window, **bool**, **bool**)]*

This table maps window managers to procedures that refine the manner in which their windows are selected and deselected. When a window is selected or deselected the table is searched for the parent window manager. If it is found the corresponding procedure is called. The first boolean parameter is **true** for a selection and **false** for a deselection. The second boolean parameter is equal to the parameter to *select* for a selection and **false** for a deselection.

*WindowMaps : **env***

This environment is intended to contain application-specific mappings from s to values, and may be added to. For example, a drawing application might provide a mapping from windows used to display objects, to the data about the corresponding objects. Each mapping is a procedure which takes a window and returns an optional *Binding*.

*browser : **proc**(Window → Optional[Binding]*

This procedure maps browser windows to the corresponding values, locations and types.

4.26.7 *Tools*

*checkBoxGroupGen : **proc**(*Appearance, ***proc**(**int**, **bool**), **int**, **int**, **int** → ChoicePack)*

This procedure generates a window displaying a group of check boxes. The interpretation of the parameters is the same as that for *genericChoiceGen*. An example of a check box window is shown in Figure 4.9:

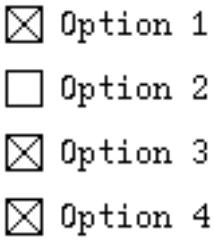


Figure 4.9: A check box window

*dialogueGen : **proc**(Size, **string**, **string**, **string**, **proc**(**int**), **proc**(**int**) → Window)*

This procedure generates a dialogue window with one or two choices. The parameters are: the size of the window; a prompt string; the titles of the two light-buttons; and the procedures to be called when the light-buttons are pressed. If the second title is empty the second light-button is not displayed. An example of a dialogue window is shown in Figure 4.10:

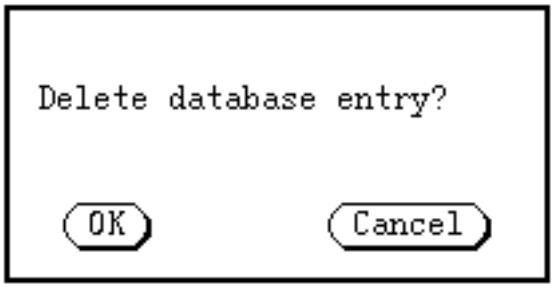


Figure 4.10: A dialogue window

---

*genericButtonGen : **proc**(Appearance, **proc**(**int**), **bool**, **proc**(**int**, **bool**), **bool** → ButtonPack)*

This procedure is used to implement *lightButtonGen* and *trillButtonGen* and it may also be called directly. The parameters are:

- a label to be displayed on the button;
- a procedure that is called when the button is pressed, passing it the number of the mouse button used;
- a boolean that specifies whether the button procedure should be called continually while the button is pressed (**true**) or whether it should only be called once the button has been released (**false**);
- a procedure that is called whenever the button changes state from pressed to released and vice-versa, passing it the number of the mouse button used and a boolean that specifies whether the button has become pressed (**true**) or released (**false**); and
- a boolean that specifies whether the button should have rounded ends (**true**) or rectangular (**false**). If an image rather than a string is supplied for the button label this parameter is ignored.

The structure returned by the generator contains a window that implements the light-button and a procedure that when called makes the light-button flash as though pressed and released. A light-button window cannot be resized.

---

*genericChoiceGen : **proc**(*Appearance, ***proc**(**int**, **bool**), **bool**, **int**, **int**, **int** → ChoicePack)*

This procedure is used to implement *checkBoxGroupGen* and *radioButtonGroupGen* and it may also be called directly. The procedure generates a group of check boxes or radio buttons. The first parameter is a vector of labels to be displayed by the boxes/buttons. The second parameter is a vector of procedures to be called when the states of the boxes/buttons are altered by clicking on them with a mouse button. The integer parameter to each procedure gives the mouse button used and the boolean parameter is **true** iff the box/button has just become selected.

The boolean parameter is **true** for check boxes and **false** for radio buttons. The integer parameters specify how the boxes/buttons are to be arranged. The first gives the number per column; the second gives the vertical separation; the third gives the horizontal separation between columns.

The states of check boxes can be altered independently of one another. The states of radio buttons are inter-dependent in that whenever a radio button becomes selected the previously selected radio button becomes de-selected.

The resulting structure contains a window on which the boxes/buttons are displayed and the following procedure:

---

*set : **proc(int, int, bool)***

This procedure sets the state of the box/button given by the first parameter to selected if the boolean parameter is **true** and unselected otherwise. The effect is the same as if the state had been set interactively by clicking with the mouse button specified by the second integer parameter.

---

*genericDialogueGen : **proc(** Size, **string**, *Appearance, ***proc(int), int, int, int, int, int** → Window)*

This procedure generates a dialogue window with an arbitrary number of choices. The parameters are: the size of the window; a prompt string; the labels for the light-buttons; the procedures to be called when the light-buttons are pressed; the horizontal offset of the message from the left side of the dialogue, the vertical offset of the bottom of the message from the top of the dialogue, the number of buttons in each column of buttons, the horizontal separation between columns of buttons and the vertical separation between rows of buttons. An example of such a dialogue window is shown in Figure 4.11:

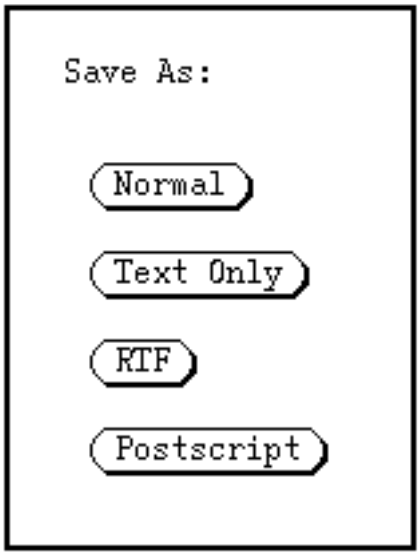


Figure 4.11: A dialogue window

---

*genericMenuExpandableGen : **proc(** *Appearance, ***proc(int**, MouseEvent), *Appearance, ***proc(int), proc(bool, bool, bool, int), bool, bool, bool** → MenuPack)*

This procedure generates a window containing a menu and associated light-buttons. The parameters are the same as those to *genericMenuGen*, except for the third and fourth parameters which specify the labels and actions for the light-buttons. An example of such a menu window is shown in Figure 4.12:

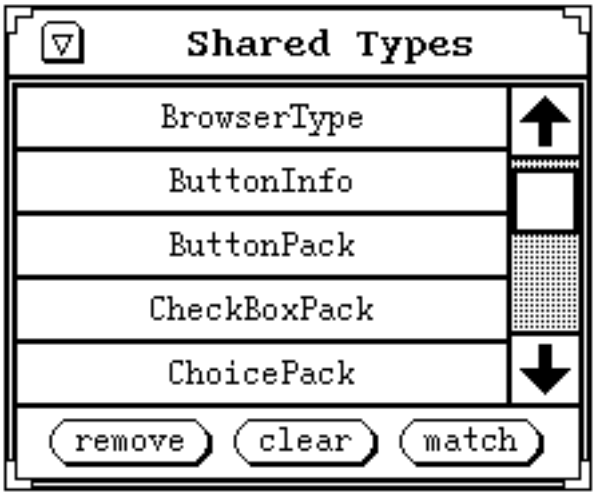


Figure 4.12: A menu window

---

*genericMenuGen : **proc(** *Appearance, *__proc(int__, MouseEvent),*
__proc(bool__, __bool__, __bool__, __int__), __bool__, __bool__, __bool__ → MenuPack)

This procedure is used to implement *menuGen*, *scrollingMenuGen* and *genericMenuExpandableGen*, and it may also be called directly. The parameters are:

- a vector containing labels for the menu entries;
- a vector containing procedures that are called whenever an entry is selected or its highlight state changes;
- a procedure that is called whenever the menu is scrolled; and
- booleans that specify whether the menu may be scrolled, whether entries may be added and removed from the menu and whether the scroll bar, if present, is on the left of the menu.

Whenever a mouse button is pressed or released over a menu entry the corresponding element of the vector of procedures is called, passing it the number of the menu entry and a description of the event type. This also occurs when the cursor moves over or leaves a menu entry while a mouse button is down.

The scroll procedure is called whenever the up or down scroll buttons are used. Its first boolean parameter is **true** when either button is pressed down initially, the second is **true** when either button is released and the third is **true** if the scrolling is in the up direction. The number of the mouse button used is also passed to it. The procedure is called continually while either scroll button is held down, in which case the first two parameters are **false**.

The resulting structure contains the menu window, a table mapping entry numbers to structures containing labels and actions, and the following procedures:

---

*setTop : **proc(int)***

This procedure scrolls the menu so that the given entry lies at or near the top of the menu, subject to the constraint that the maximum possible number of entries for the current window size are always displayed. Thus if the given

entry is the last it will never be displayed higher than the bottom position. If the menu is non-scrollable then the procedure has no effect.

---

getTop : ***proc(→ int)***

This procedure returns the number of the entry currently displayed at the top of the menu.

---

setNoVisible : ***proc(int)***

This procedure sets the size of the menu window so that the given number of entries are visible. The procedure has no effect if the given number is less than one. The change in size is subject to the menu window's usual constraints on minimum and maximum size.

---

getNoVisible : ***proc(→ int)***

This procedure returns the number of entries currently visible in the menu.

---

setHighlight : ***proc(int, bool)***

This procedure sets the highlighted state of the given entry to on or off depending on the boolean parameter. A value of **true** gives a highlighted entry and **false** a non-highlighted entry.

---

getHighlight : ***proc(int → bool)***

This procedure returns the highlighted state of the given entry.

---

getNoEntries : ***proc(→ int)***

This procedure returns the number of entries currently in the menu.

---

genericSliderGen : ***proc(int, int, real, real, real, bool, proc(int, real),***
proc(int, bool) → SliderPack)

This procedure is used to implement *sliderGen* and it may also be called directly. The parameters are:

- the X and Y dimensions of the slider;
- the minimum and maximum real values associated with the slider;
- the amount to be skipped when a mouse click occurs off the thumb;
- a boolean that specifies whether the slider is oriented horizontally (**true**) or vertically (**false**);
- a procedure that is called when the value of the slider is changed, passing it the mouse button used and the new value;

- a procedure that is called at the start and finish of a change in the value of the slider. When the value starts to change it is passed the mouse button used and the value **true**. When the value finishes changing it is passed the mouse button used and the value **false**.

The resulting structure contains the slider window and the following procedures:

---

*set : **proc(real)***

This procedure sets the value of the slider.

---

*setBounds : **proc(real**, **real**, **real)***

This procedure adjusts the minimum and maximum values of the slider and the skip increment.

---

*lightButtonGen : **proc**(Appearance, **proc(int)** → ButtonPack)*

This procedure generates a light-button with the given label. When the button is pressed it is highlighted. When it is released the given procedure is called, passing it the mouse button used. The structure returned contains the light-button window and a procedure that when called makes the light-button flash as though pressed and released. A light-button window cannot be resized. An example of a light-button window is shown in Figure 4.13:

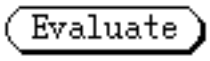


Figure 4.13: A light-button window

---

*menuGen : **proc**(*Appearance, ***proc(int)** → MenuPack)*

This procedure generates a fixed size, non-scrollable menu with the given labels and associated actions. The fields of the resulting structure are described above for *genericMenuGen*. An example of a menu window is shown in Figure 4.14:

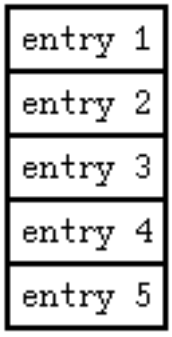


Figure 4.14: A menu window

---

*radioButtonGroupGen : **proc**(*Appearance, ***proc(int**, **bool**), **int**, **int**, **int** → ChoicePack)*

This procedure generates a group of radio buttons. The interpretation of the parameters is the same as that described for *genericChoiceGen* above. An example of a radio button window is shown in Figure 4.15:

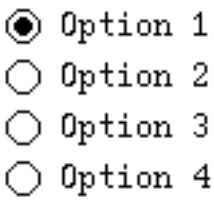


Figure 4.15: A radio button window

---

*scrollingMenuGen : **proc**(*Appearance, ***proc**(**int**) → MenuPack)*

This procedure generates a variable size scrollable menu with the given labels and associated actions. The fields of the resulting structure are described above for *genericMenuGen*. Entries may not be added to or removed from the menu. The number of entries initially visible is not defined. An example of a scrolling menu window is shown in Figure 4.16:

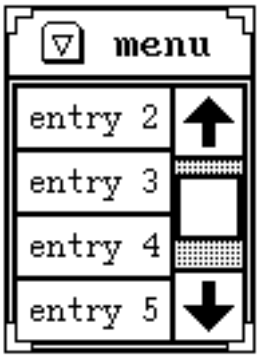


Figure 4.16: A scrolling menu window

---

*sliderGen : **proc**(**int**, **int**, **real**, **real**, **real**, **bool**, **proc**(**int**, **real**) → SliderPack)*

This procedure generates a slider. The interpretation of the parameters is the same as that described for *genericSliderGen* above. An example of a slider window is shown in Figure 4.17:

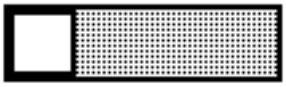


Figure 4.17: A slider window

---

*trillButtonGen : **proc**(Appearance, **proc**(**int**) → ButtonPack)*

This procedure generates a light-button with the given label. When the button is pressed it is highlighted and the given procedure is called repeatedly, passing it the mouse button used, until the button is released. The structure returned contains the light-button window and a procedure that when called makes the light-button flash as though pressed and released. The light-button window cannot be resized.

*copyClearEditorToolGen : **proc**[HyperLink](Size, Editor[HyperLink] → EditorPack[HyperLink])*

This procedure generates an editor pack that provides buttons for copying and clearing the hyper-text. The parameters are the size of the editor window and an editing buffer. An example of such an editor window is shown in Figure 4.18:

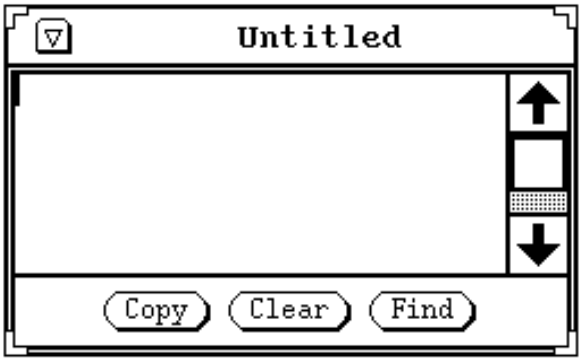


Figure 4.18: An editor window with *copy* and *clear*

*fullEditorToolExpandableGen : **proc**[HyperLink](Size, Editor[HyperLink], ***string**, ***proc(int**) → EditorPack[HyperLink])*

This procedure generates an editor pack that provides all the buttons described for *genericEditorToolGen* and also user defined buttons. The parameters are the size of the editor window, an editing buffer, a vector of string labels for the user defined buttons and a vector of associated actions. An example of such an editor window is shown in Figure 4.19:

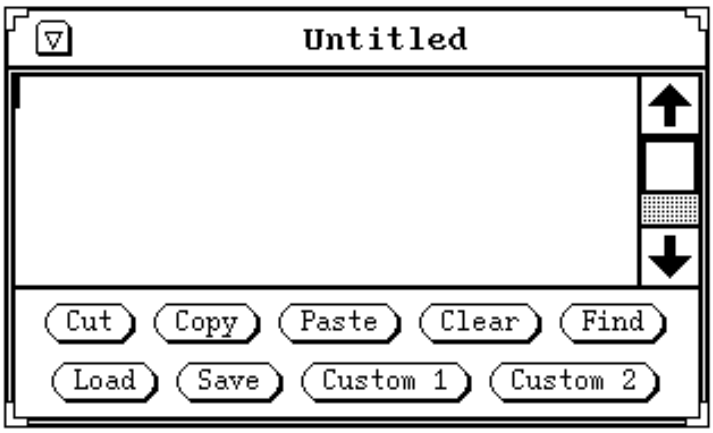


Figure 4.19: An editor window with all operations and user defined buttons

*fullEditorToolGen : **proc**[HyperLink](Size, Editor[HyperLink] → EditorPack[HyperLink])*

This procedure generates an editor pack that provides all the buttons described for *genericEditorToolGen*. The parameters are the size of the editor window and an editing buffer. An example of such an editor window is shown in Figure 4.20:

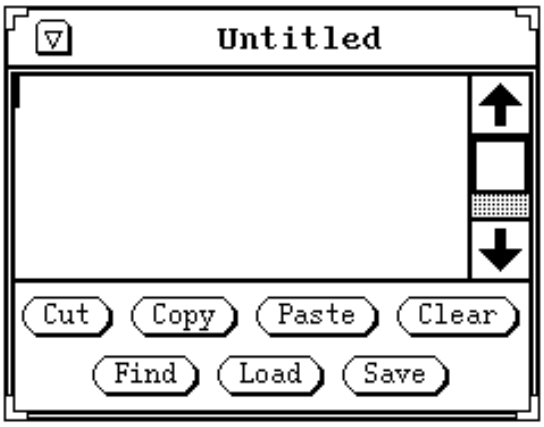


Figure 4.20: An editor window with all operations

---

*genericEditorToolGen : **proc**[HyperLink](Size, Editor[HyperLink], **bool**, **bool**, **bool**, **bool**, **bool**, **bool**, *__string__, *__proc__(**int**)*
→ EditorPack[HyperLink])

This procedure is used to implement the other editor generators and it may also be called directly. The parameters are the size of the editor window, an editing buffer, boolean parameters described below, a vector of string labels for user defined buttons and a vector of actions for the user defined actions. The boolean parameters specify whether the following options are enabled, respectively:

- ability to select hyper-text interactively using the mouse;
- ability to copy selected hyper-text into the given buffer;
- ability to clear the hyper-text in the editor;
- ability to edit the hyper-text interactively;
- ability to load and save text from and to the file system; and
- presence of a scroll bar.

The number of light-buttons displayed at the bottom of the editor window depends on which of the options are enabled. The possible buttons are *cut*, *copy*, *paste*, *clear*, *find*, *load* and *save*. The dependencies are shown in Table 4.4, in which a tick indicates that a particular option must be enabled for the corresponding button to appear:

		option enabled				
		select	*copy*	*clear*	*edit*	*load/save*
button	*cut*		✔		✔	
	copy		✔			
	paste		✔		✔	
	clear			✔		
	find		✔			
	load					✔
	save				✔	✔

Table 4.4: Editor tool light-buttons

User defined buttons are displayed after the pre-defined buttons. The resulting structure contains the editor window, the editor and the following procedures:

---

getTitle : ***proc****(→ **string**)*

This procedure returns the current title of the text being edited. This title corresponds to the most recent file name if text has been loaded from or saved to the file system.

---

getText : ***proc****(→ HyperText[HyperLink])*

This procedure returns the current contents of the editor.

---

append : ***proc****(HyperText[HyperLink])*

This procedure inserts the given hyper-text at the end of the existing hyper-text.

---

genericSingleLineDialogueGen : ***proc****[HyperLink](Size,* ***string****, *Appearance,*
*****proc****(**int**, HyperText[HyperLink]),*
EditorPack[HyperLink], ***int****,* ***int****,* ***int****,*
int*,* ***int****,* ***int****,* ***int*** *→*
DialoguePack[HyperLink])

This procedure is used to implement *singleLineDialogueGen* and it may also be called directly. It generates a dialogue window containing a single line hyper-text editor and a number of user defined light-buttons. The parameters are as follows:

- the size of the dialogue window;
- a prompt label;
- a vector of labels for the light-buttons;

- a vector of procedures to be called when the light-buttons are pressed, each one taking as parameters the mouse button used and the current contents of the editor;
- the editor to be displayed;
- the X offset of the prompt from the left of the dialogue; the Y offset of the prompt from the top of the dialogue; the X offset of the editor from the left of the dialogue; the Y offset of the editor from top of the dialogue; the number of light-buttons per column; the horizontal separation of the light-buttons; and the vertical separation of the light-buttons.

The resulting structure contains the dialogue window and the following procedure:

---

set : ***proc****(HyperText[HyperLink])*

This procedure replaces the contents of the editor with the given hyper-text.

---

readOnlyEditorToolExpandableGen : ***proc****[HyperLink](Size, Editor[HyperLink],* ****string****,*
****proc****(* ***int*** *) →*
EditorPack[HyperLink])

This procedure generates an editor pack with user defined buttons that does not allow interactive selecting, copying or editing of the hyper-text. The parameters are the size of the editor window, an editing buffer, a vector of string labels for the user defined buttons and a vector of associated actions. An example of such an editor window is shown in Figure 4.21:

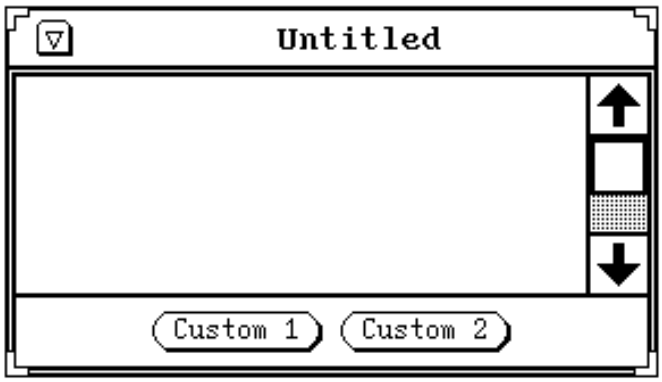


Figure 4.21: A read only editor window with user defined buttons

---

readOnlyEditorToolGen : ***proc****[HyperLink](Size, Editor[HyperLink] →*
EditorPack[HyperLink])

This procedure generates an editor pack that does not allow interactive selecting, copying or editing of the hyper-text. The parameters are the size of the editor window and an editing buffer. An example of such an editor window is shown in Figure 4.22:

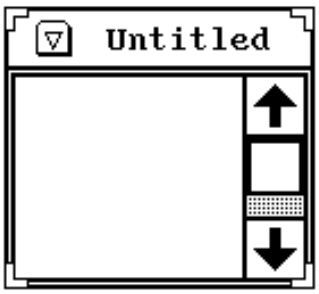


Figure 4.22: A read only editor window

---

*simpleEditorToolGen : **proc**[HyperLink](Size, Editor[HyperLink]*
→ EditorPack[HyperLink])

This procedure generates an editor pack that allows interactive selecting and editing of the hyper-text but provides no light-buttons. The parameters are the size of the editor window and an editing buffer. An example of such an editor window is shown in Figure 4.23:

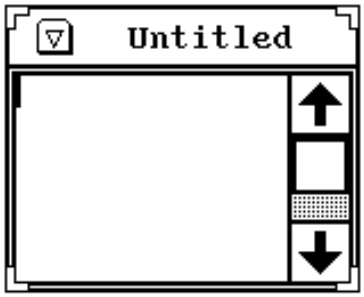


Figure 4.23: A simple editor window

---

*singleLineDialogueGen : **proc**[HyperLink](Size, **string**, **string**, **string**,*
__proc__(HyperText[HyperLink]),
__proc__(HyperText[HyperLink]) →
DialoguePack[HyperLink])

This procedure generates a dialogue window containing a single line hyper-text editor and two user defined light-buttons. The parameters are: the size of the dialogue; a prompt label; labels for the light-buttons; and procedures that are called when the light-buttons are pressed, passing them the current contents of the editor. An example of such a dialogue window is shown in Figure 4.24:

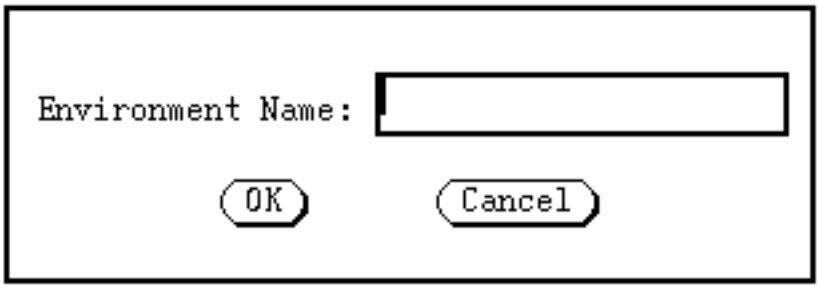


Figure 4.24: A single line dialogue window

4.26.8 *Utilities*

---

*button1Down,button2Down,button3Down : **proc**(Event → **bool**)*

These procedures return **true** iff the corresponding mouse button of the given event is down.

---

*changeCursor : **proc**(**string**)*

This procedure sets the cursor image to the image denoted by the given string. The cursors available are "border", "iconise", "move", "resize" and "wait".

---

*fileToImage : **proc**(**file** → **image**)*

This procedure reads a representation of an image from the given file in the format produced by *imageToFile* and converts it to an image.

---

*fileToSound : **proc**(**file** → ***int**)*

This procedure reads a representation of a sound from the given file in Sun audio format and converts it to a vector of integers.

---

*getBorderExtent : **proc**(Window → Rect)*

This procedure returns the extent of the window's current border. The coordinates of the corners are given relative to the origin of the window.

---

*imageToFile : **proc**(**image**, **file**)*

This procedure writes a representation of the given image to the given file.

---

*playSound : **proc**(***int**)*

This procedure attempts to play the sound represented by the given vector through the machine's loudspeaker.

---

*setColourMap : **proc**(***int**, **int**)*

This procedure sets the Napier system colour map to the given map. The second parameter specifies the screen depth in planes.

---

*tiffFileToImage : **proc**(**file** → Pair[**image**, ***int**])*

This procedure reads a representation of an image from the given file in 1-bit, 4-bit or 8-bit TIFF format and converts it to an image and an associated colour map. The procedure was written by Ying-Jean Kuo of Glasgow University.

5 The *Error* Environment

The *Error* environment contains the following items:

Arithmetic :	**env**
Environment :	**env**
Format :	**env**
Graphics :	**env**
IO :	**env**
String :	**env**
Structure :	**env**
Variant :	**env**
Vector :	**env**

The procedures that are called in the event of an error are stored in these environments. Each procedure is a variable and the user may change them by assignment. By default, all the error procedures write out an appropriate error message and halt the execution of the current thread.

5.1 *Arithmetic* Errors

---

unaryInt : ***proc(string, int → int)***

This procedure may be called during the operations *unary minus* and *abs* with a string parameter of "-" and "abs" respectively. The second parameter is the integer on which the original operation was attempted.

Int : ***proc(string, int, int → int)***

This procedure may be called during the operations *plus*, *times*, *minus*, *div* and *rem* with a string parameter of "+", "*", "-", "div" and "rem" respectively. The other parameters are the integers on which the original operation was attempted.

---

unaryReal : ***proc(string, real → real)***

This procedure may be called during the operations *unary minus*, *sin*, *cos*, *exp*, *ln*, *sqrt*, *atan* and *truncate* with a string parameter of "-", "sin", "cos", "exp", "ln", "sqrt", "atan" and "truncate" respectively. The second parameter is the real number on which the original operation was attempted.

Real : ***proc(string, real, real → real)***

This procedure may be called during the operations *plus*, *times*, *minus* and *divide* with a string parameter of "+", "*", "-" and "/" respectively. The other parameters are the real numbers on which the original operation was attempted.

*getByte : **proc(int, int → int)***

This procedure may be called during execution of the *getByte* procedure in *PrimitiveIO*. It is called when the byte index supplied to *getByte* is not between 0 and 3. The parameters to the error procedure are those supplied to the original call of *getByte*.

---

*setByte : **proc(int, int, int → int)***

This procedure may be called during execution of the *setByte* procedure in *PrimitiveIO*. It is called when the byte index supplied to *setByte* is not between 0 and 3. The parameters to the error procedure are those supplied to the original call of *setByte*.

---

*truncate : **proc(real → int)***

This procedure may be called during execution of the *truncate* procedure in *Arithmetic*. It is called when the result would be outwith the implementation dependent bounds for legal integers. The parameter to the error procedure is that supplied to the original call of *truncate*.

5.2 *Graphics* Errors

---

*Draw : **proc(pic, real, real, real, real)***

This procedure may be called during execution of the procedure returned by the *makeDrawFunction* procedure in *Outline*. It is called when the two *x* parameters or the two *y* parameters are equal. The parameters to the error procedure are the picture to be drawn and the attempted clipping region.

---

*Text : **proc(string, real, real, real, real → pic)***

This procedure may be called during creation of a picture using the **text** statement. It is called when the end points of the text are coincident. The parameters to the error procedure are the text string, the coordinates of the first end point and the coordinates of the second end point.

---

*getPixel : **proc(image, int, int → pixel)***

This procedure may be called during execution of the *getPixel* procedure in *Raster*. It is called when the coordinates lie outwith the bounds of the image. The parameters to the error procedure are those supplied to the original call of *getPixel*.

---

*setPixel : **proc(image, int, int, pixel)***

This procedure may be called during execution of the *setPixel* procedure in *Raster*. It is called when the coordinates lie outwith the bounds of the image. The parameters to the error procedure are those supplied to the original call of *setPixel*.

---

pixelOverflow : ***proc(pixel → pixel)***

This procedure may be called during the pixel concatenation operation ++. It is called when the depth of the resulting pixel overflows the implementation size (the maximum pixel depth is 24 pixels in the current implementation of Napier88). The parameter to the error procedure is a pixel containing the first 24 planes of the result.

---

subPixel : ***proc(pixel, int, int → pixel)***

This procedure may be called during the pixel indexing operation |. It is called when the start plane is less than zero, when the start plane is greater than or equal to the pixel depth, when the number of planes requested is less than one, or when the planes selected are not a subset of the original pixel. The parameters to the error procedure are the original pixel, the start plane and the number of planes.

---

makeImage : ***proc(int, int, pixel → pixel)***

This procedure may be called during the image creation operation. It is called when either the *x* or the *y* dimension is less than one. The parameters to the error procedure are the *x* and *y* dimensions and the initialising pixel.

---

subImage : ***proc(image, int, int → image)***

This procedure may be called during the image indexing operation |. It is called when the start plane is less than zero, when the start plane is greater than or equal to the image depth, when the number of planes requested is less than one, or when the planes selected are not a subset of the original image. The parameters to the error procedure are the original image, the start plane and the number of planes.

---

limitAt : ***proc(image, int, int → image)***

This procedure may be called during the '**limit** i **at** x, y' operation. It is called when $x < 0$ or $x \geq xDim$ (i) or $y < 0$ or $y \geq yDim$ (i). The parameters to the error procedure are the original image and the *x* and *y* coordinates.

---

limitAtBy : ***proc(image, int, int, int, int → image)***

This procedure may be called during the '**limit** i **to** x1 **by** y1 **at** x2, y2' operation. It is called when $x2 < 0$ or $x2 \geq xDim$ (i) or $y2 < 0$ or $y2 \geq yDim$ (i) or when the sub-image requested is not totally enclosed within the original image. The parameters to the error procedure are the original image, the *x* coordinate, the *x* dimension, the *y* coordinate and the *y* dimension.

---

imagePixelConstant : ***proc(image)***

This procedure is called when a raster update operation is attempted on an image of constant pixels. The parameter to the error procedure is the original image.

---

*getScreen : **proc(file → image)***

This procedure may be called during execution of the *getScreen* procedure in *Device*. It is called when the file is not a raster device. The parameter to the error procedure is the original file.

---

*locator : **proc(file, *int)***

This procedure may be called during execution of the *locator* procedure in *Device*. It is called when the file is not a mouse or tablet device. The parameters to the error procedure are those supplied to the original call of *locator*.

---

*colourMap : **proc(file, pixel, int)***

This procedure may be called during execution of the *colourMap* procedure in *Device*. It is called when the file is not a raster device. The parameters to the error procedure are those supplied to the original call of *colourMap*.

---

*colourOf : **proc(file, pixel → int)***

This procedure may be called during execution of the *colourOf* procedure in *Device*. It is called when the file is not a raster device. The parameters to the error procedure are those supplied to the original call of *colourOf*.

---

*getCursor : **proc(file → image)***

This procedure may be called during execution of the *getCursor* procedure in *Device*. It is called when the file is not a raster device. The parameters to the error procedure are those supplied to the original call of *getCursor*.

---

*setCursor : **proc(file, image)***

This procedure may be called during execution of the *setCursor* procedure in *Device*. It is called when the file is not a raster device. The parameters to the error procedure are those supplied to the original call of *setCursor*.

---

*getCursorInfo : **proc(file, *int)***

This procedure may be called during execution of the *getCursorInfo* procedure in *Device*. It is called when the file is not a raster device. The parameters to the error procedure are those supplied to the original call of *getCursorInfo*.

---

*setCursorInfo : **proc(file, *int)***

This procedure may be called during execution of the *setCursorInfo* procedure in *Device*. It is called when the file is not a raster device. The parameters to the error procedure are those supplied to the original call of *setCursorInfo*.

*closedWindow : **proc(file, image)***

This procedure may be called during a raster operation on an image associated with a window file. It is called when the window file is closed. The parameters to the error procedure are the window file and the image.

5.3 *String* Errors

*concatenate : **proc(string, string → string)***

This procedure may be called during the string concatenation operation ++. It is called when the length of the resulting string overflows the implementation size (the maximum string length is *maxint* characters in the current implementation of Napier88). The parameters to the error procedure are the two strings to be concatenated.

*subString : **proc(string, int, int → string)***

This procedure may be called during the substring indexing operation |. It is called when the string to be dereferenced is an empty string, when the start position is less than one, when the length is less than zero, or when the finish position is after the end of the string. The parameters to the error procedure are the original string, the start position and the number of characters.

5.4 *Structure* Errors

*structureFieldConstant : **proc()***

This procedure may be called during an assignment to a field within a structure. It is called when the field is constant.

5.5 *Vector* Errors

*vectorElementConstant : **proc**[t](*t, **int**, t)*

This procedure may be called during an assignment to a location within a vector. It is called when the elements of the vector are constant. The parameters to the error procedure are the original vector, the index and the value to be assigned to the location.

*vectorIndexAssign : **proc**[t](*t , **int**, t)*

This procedure may be called during an attempted assignment to a location within a vector. It is called when the attempted index is less than the lower bound of the vector or greater than the upper bound of the vector. The parameters to the error procedure are the original vector, the index and the value to be assigned to the location.

*vectorIndexSubs : **proc**[t](*t, **int** → t)*

This procedure may be called during an attempted access to a location within a vector. It is called when the attempted index is less than the lower bound of the vector or greater than the upper bound of the vector. The parameters to the error procedure are the original vector and the index.

---

*makeVector : **proc**[t](**int**, **int**, t → *t)*

This procedure may be called during an attempted vector creation. It is called when the lower bound is greater than the upper bound. The parameters to the error procedure are the lower bound, the upper bound and the initialising value.

5.6 *Variant* Errors

---

*varProject : **proc**(TypeRep, **string**, **string**)*

This procedure may be called during a variant projection. It is called when the variant is projected onto an incorrect branch. The parameters to the error procedure are a representation of the variant type, the name of the expected branch and the name of the actual branch.

The Napier88 system cannot continue after a variant projection error and the current thread will terminate even if the error handling procedure returns normally.

5.7 *Environment* Errors

---

*envRedeclaration : **proc**(**env**, **string**, TypeRep, **bool**)*

This procedure may be called during a declaration within an environment. It is called when an attempt is made to declare a new binding with an identifier already present in the environment. The parameters to the error procedure are the original environment, the name of the identifier, a representation of the type of the new binding and a boolean that is **true** iff the new binding was to be constant.

---

*envProject : **proc**(**env**, **string**, TypeRep, **bool**)*

This procedure may be called during a projection from an environment. It is called when no binding with the required signature can be found in the environment. The parameters to the error procedure are the original environment, the name of the identifier being searched for, a representation of the expected type and a boolean that is **true** iff the binding was expected to be constant.

---

*envDrop : **proc**(**env**, **string**)*

This procedure may be called during the dropping of a binding from an environment. It is called when no binding with the required identifier can be found in the environment. The parameters to the error procedure are the original environment and the name of the identifier that was to be dropped.

5.8 *IO* Errors

---

writeByte : ***proc(file, int, int)***

This procedure may be called during execution of the *writeByte* procedure in *IO*. It is called when an error occurs. The parameters to the error procedure are the file being written to, the byte being written and the I/O number indicating the error.

---

writeString : ***proc(file, string, int, int)***

This procedure may be called during execution of the *writeString* procedure in *IO*. It is called when an error occurs. The parameters to the error procedure are the file being written to, the string being written, the number of characters successfully written and the I/O number indicating the error.

---

writeBool : ***proc(file, string, int, int)***

This procedure may be called during execution of the *writeBool* procedure in *IO*. It is called when an error occurs. The parameters to the error procedure are the file being written to, a string representation of the boolean being written, the number of characters successfully written and the I/O number indicating the error.

---

writeInt : ***proc(file, string, int, int)***

This procedure may be called during execution of the *writeInt* procedure in *IO*. It is called when an error occurs. The parameters to the error procedure are the file being written to, a string representation of the integer being written, the number of characters successfully written and the I/O number indicating the error.

---

writeReal : ***proc(file, string, int, int)***

This procedure may be called during execution of the *writeReal* procedure in *IO*. It is called when an error occurs. The parameters to the error procedure are the file being written to, a string representation of the real being written, the number of characters successfully written and the I/O number indicating the error.

---

endOfInputIOE : ***proc(file, int → bool)***

This procedure may be called during execution of the *endOfInput* procedure in *IO*. It is called when an error occurs. The parameters to the error procedure are the file being read and the I/O number indicating the error.

---

readByteIOE : ***proc(file, int → int)***

This procedure may be called during execution of the *readByte* procedure in *IO*. It is called when an error occurs. The parameters to the error procedure are the file being read and the I/O number indicating the error.

---

readByteEOI : ***proc(file → int)***

This procedure may be called during execution of the *readByte* procedure in *IO*. It is called when the end of input is encountered. The parameter to the error procedure is the file being read.

---

readCharIOE : ***proc(file, int → string)***

This procedure may be called during execution of the *readChar* procedure in *IO*. It is called when an error occurs. The parameters to the error procedure are the file being read and the I/O number indicating the error.

---

readCharEOI : ***proc(file → string)***

This procedure may be called during execution of the *readChar* procedure in *IO*. It is called when the end of input is encountered. The parameter to the error procedure is the file being read.

---

peekByteIOE : ***proc(file, int → int)***

This procedure may be called during execution of the *peekByte* procedure in *IO*. It is called when an error occurs. The parameters to the error procedure are the file being read and the I/O number indicating the error.

---

peekByteEOI : ***proc(file → int)***

This procedure may be called during execution of the *peekByte* procedure in *IO*. It is called when the end of input is encountered. The parameter to the error procedure is the file being read.

---

endOfInputUnread : ***proc(file, int, int → int)***

This procedure may be called during execution of the *endOfInput* procedure in *IO*. It is called when the byte read cannot be made available to the next read operation on the same file. The parameters to the error procedure are the file being read, the byte read and the I/O number indicating the error.

---

peekByteUnread : ***proc(file, int, int → int)***

This procedure may be called during execution of the *peekByte* procedure in *IO*. It is called when the byte read cannot be made available to the next read operation on the same file. The parameters to the error procedure are the file being read, the byte read and the I/O number indicating the error.

---

peekCharIOE : ***proc(file, int → string)***

This procedure may be called during execution of the *peekChar* procedure in *IO*. It is called when an error occurs. The parameters to the error procedure are the file being read and the I/O number indicating the error.

---

peekCharEOI : ***proc(file → string)***

This procedure may be called during execution of the *peekChar* procedure in *IO*. It is called when the end of input is encountered. The parameter to the error procedure is the file being read.

---

peekCharUnread : ***proc(file, int, int → string)***

This procedure may be called during execution of the *peekChar* procedure in *IO*. It is called when the character read cannot be made available to the next read operation on the same file. The parameters to the error procedure are the file being read, the byte corresponding to the character read and the I/O number indicating the error.

---

readBoolIOE : ***proc(file, string, int → bool)***

This procedure may be called during execution of the *readBool* procedure in *IO*. It is called when an error occurs. The parameters to the error procedure are the file being read, the characters that had been read when the error occurred, excluding any layout characters, and the I/O number indicating the error.

---

readBoolEOI : ***proc(file, string → bool)***

This procedure may be called during execution of the *readBool* procedure in *IO*. It is called when the end of input is encountered. The parameters to the error procedure are the file being read and the characters that had been read when the end of input was detected, excluding any layout characters.

---

readBoolBadChar : ***proc(file, string → bool)***

This procedure may be called during execution of the *readBool* procedure in *IO*. It is called when an erroneous character is read. The parameters to the error procedure are the file being read and the characters that had been read, up to and including the erroneous character, excluding any layout characters.

---

readStringIOE : ***proc(file, string, int → string)***

This procedure may be called during execution of the *readString* procedure in *IO*. It is called when an error occurs. The parameters to the error procedure are the file being read, the characters that had been read when the error occurred, excluding the leading double quote, and the I/O number indicating the error.

---

readStringEOI : ***proc(file, string → string)***

This procedure may be called during execution of the *readString* procedure in *IO*. It is called when the end of input is encountered. The parameters to the error procedure are the file being read and the characters that had been read when the end of input was detected, excluding the leading double quote.

*readStringBadChar : **proc(file, string → string)***

This procedure may be called during execution of the *readString* procedure in *IO*. It is called when a double quote character is not the first non-layout character found. The parameters to the error procedure are the file being read and the character that was read instead of the expected double quote.

---

*readLineIOE : **proc(file, string, int → string)***

This procedure may be called during execution of the *readLine* procedure in *IO*. It is called when an error occurs. The parameters to the error procedure are the file being read, the characters that had been read when the error occurred and the I/O number indicating the error.

---

*readLineEOI : **proc(file, string → string)***

This procedure may be called during execution of the *readLine* procedure in *IO*. It is called when the end of input is encountered. The parameters to the error procedure are the file being read and the characters that had been read when the end of input was detected.

---

*readIntIOE : **proc(file, string, int → int)***

This procedure may be called during execution of the *readInt* procedure in *IO*. It is called when an error occurs. The parameters to the error procedure are the file being read, the characters that had been read when the error occurred, excluding any layout characters, and the I/O number indicating the error.

---

*readIntEOI : **proc(file, string → int)***

This procedure may be called during execution of the *readInt* procedure in *IO*. It is called when the end of input is encountered. The parameters to the error procedure are the file being read and the characters that had been read when the end of input was detected, excluding any layout characters.

---

*readIntBadChar : **proc(file, string → int)***

This procedure may be called during execution of the *readInt* procedure in *IO*. It is called when a digit or a sign character followed by a digit is not the first non-layout character found. The parameters to the error procedure are the file being read and the character that was read instead of the expected digit or sign character.

---

*readIntUnread : **proc(file, string, int, int → int)***

This procedure may be called during execution of the *readInt* procedure in *IO*. It is called when an extra character read while parsing an integer cannot be made available to the next read operation on the same file. The parameters to the error procedure are the file being read, the characters that form the integer, the byte corresponding to the extra character read and the I/O number indicating the error.

---

readIntOverflow : ***proc(file, string → int)***

This procedure may be called during execution of the *readInt* procedure in *IO*. It is called when an arithmetic error occurs converting the integer parsed into an integer value. The parameters to the error procedure are the file being read and the characters that form the integer.

---

readRealIOE : ***proc(file, string, int → real)***

This procedure may be called during execution of the *readReal* procedure in *IO*. It is called when an error occurs. The parameters to the error procedure are the file being read, the characters that had been read when the error occurred, excluding any layout characters, and the I/O number indicating the error.

---

readRealEOI : ***proc(file, string → real)***

This procedure may be called during execution of the *readReal* procedure in *IO*. It is called when the end of input is encountered. The parameters to the error procedure are the file being read and the characters that had been read when the end of input was detected, excluding any layout characters.

---

readRealBadChar : ***proc(file, string → real)***

This procedure may be called during execution of the *readReal* procedure in *IO*. It is called when a digit or a sign character followed by a digit is not the first non-layout character found. The parameters to the error procedure are the file being read and the character that was read instead of the expected digit or sign character.

---

readRealUnread : ***proc(file, string, int, int → real)***

This procedure may be called during execution of the *readReal* procedure in *IO*. It is called when an extra character read while parsing a real cannot be made available to the next read operation on the same file. The parameters to the error procedure are the file being read, the characters that form the real, the byte corresponding to the extra character read and the I/O number indicating the error.

---

readRealOverflow : ***proc(file, string → real)***

This procedure may be called during execution of the *readReal* procedure in *IO*. It is called when an arithmetic error occurs converting the real parsed into a real value. The parameters to the error procedure are the file being read and the characters that form the integer.

5.9 *Format* Errors

---

fformat : ***proc(real, int, int → string)***

This procedure may be called during execution of the *fformat* procedure in *Format*. It is called when the real number is too large to be written with the required number of digits before the point, or when the required number of digits after the point is negative. The

parameters to the error procedure are the real number and the required numbers of digits before and after the point.

---

eformat : ***proc(real, int, int → string)***

This procedure may be called during execution of the *eformat* procedure in *Format*. It is called when either of the required number of digits before and after the point is negative. The parameters to the error procedure are the real number and the required numbers of digits before and after the point.

6 Type Definitions

The Napier88 types required for programming using WIN are listed below. They are contained in the declaration set *userTypes*.

6.1 General

```
type Pos is         structure( x,y : int )
type Size is        structure( x,y : int )
type Limit is       structure( pos : Pos ; size : Size )
type Rect is        structure( origin,corner : Pos )
type Level is       structure( fromFront : bool ; position : int )
type InputOption is variant( all,none,normal : null )

type Optional[ T ] is variant( present : T ; absent : null )

rec type List[ T ] is variant( cons : structure( hd : T ; tl : List[ T ] ) ; tip : null )

rec type DoubleList[ T ] is variant(
     cons : structure( hd : T ; before,after : DoubleList[ T ] ) ; tip : null )

type Pair[ S,T ] is structure( fst : S ; snd : T )
```

6.2 Event Distribution

```
type Mouse is structure( x,y : int ; buttons : *bool )
type Event is variant( chars : string;
                       mouse : Mouse;
                       select,deselect : null )
type EventType is variant( up,down,enter,leave,click,doubleClick : null )
type MouseEvent is structure( button : int ; event : EventType )

type Application is proc( Event )

type EventTest is proc( Event -> bool )

type Notification is structure( examineEvent : EventTest ; processEvent : Application )

type Notifier is structure( distributeEvent : Application;
                            addNotification : proc( Notification,Level -> proc() ) )
```

6.3 Windows and Window Managers

```
type ResizeControl is structure( before : proc( Rect -> Rect ) ; after : proc( Rect ) )

rec type DisplayInfo is structure( window : Window ; pos : Pos ; level : Level )

& Window is structure( windowRaster :     proc( Limit,Limit,Window,int,bool );
                       imageRaster :      proc( Limit,image,int,bool );
                       drawLine :         proc( Pos,Pos,pixel,int );
                       setInputOption :   proc( InputOption );
                       getInputOption :   proc( -> InputOption );
                       setSize :          proc( Rect );
                       getSize :          proc( -> Size );
                       setApplication :   proc( Application );
                       getApplication :   proc( -> Application );
                       setTitle :         proc( string );
                       getTitle :         proc( -> string );
                       setResizeControl : proc( ResizeControl );
                       getResizeControl : proc( -> ResizeControl );
```

```
                        setMinSize :       proc( Size );
                        getMinSize :       proc( -> Size );
                        setMaxSize :       proc( Size );
                        getMaxSize :       proc( -> Size );
                        setDepth :         proc( int );
                        getDepth :         proc( -> int );
                        setBorderStyle :   proc( BorderStyle );
                        getBorderStyle :   proc( -> BorderStyle );
                        getBorder :        proc( -> Border );
                        setCursor :        proc( image );
                        getCursor :        proc( -> image );
                        getWindowManager : proc( -> WindowManager );
                        setVirtualWindow : proc( string, Window ) )

& WindowManager is structure( display :          proc( DisplayInfo,bool );
                              undisplay :        proc( Window );
                              makeCurrent :      proc( Window );
                              setPos :           proc( Window,Pos );
                              getPos :           proc( Window -> Pos );
                              setLevel :         proc( Window,Level );
                              getLevel :         proc( Window,bool -> Level );
                              getWindows :       proc( -> *Window );
                              getWindowAtPos :   proc( Pos -> Window );
                              getNotifier :      proc( string -> Notifier );
                              getDisplayWindow : proc( -> Window );
                              getIconManager :   proc( -> IconManager );
                              setBackgroundApp : proc( Application );
                              getBackgroundApp : proc( -> Application ) )

& IconManager is structure( close :          proc( Window );
                            open :           proc( Window );
                            getIconState :   proc( Window -> DisplayInfo );
                            getWindowState : proc( Window -> DisplayInfo ) )

& BorderStyle is proc( Window -> Border )
& Border is      List[ Area ]
& Area is        structure( currentImage,nonCurrentImage : image ;
                            pos : Pos ; distributeEvent : Application )
```

6.4 Fonts, Tables and Lists

```
type Font is structure( characters : *image ; fontHeight,descender : int ; info : string )
type FontPack is structure( font : Font ; stringToTile,charToTile : proc( string -> image ) )

type Table[ Key,Data ] is structure( enter :  proc( Key,Data );
                                     lookup : proc( Key -> Optional[ Data ] );
                                     remove : proc( Key );
                                     scan :   proc( proc( Key,Data -> bool ) ) )

type Comparison[ Key ] is variant(
     ordered :   structure( equal,lessThan : proc( Key,Key -> bool ) );
     unordered : structure( equal : proc( Key,Key -> bool ) ) )

type ListPack[ T ] is structure( insert,
                                 replace : proc( T );
                                 clear,
                                 delete :  proc();
                                 element : proc( -> Optional[ T ] );
                                 length,
                                 pos :     proc( -> int );
                                 atEnd :   proc( -> bool );
                                 go :      proc( int );
                                 goNext,
                                 goPrev :  proc();
                                 find :    proc( proc( T -> bool ) -> int ) )
```

6.5 Editors

```
type Index is variant( characters : int;
                       lines :      structure( line,char : int ) )

type CodeRegion is structure( start,finish : int )

type Substitution[ T ] is structure( value : T ; region : CodeRegion )

type LinkPack[ HyperLink ] is structure( link :     HyperLink ;
                                         showLink : proc( HyperLink,int ) )

type HyperText[ HyperLink ] is structure(
     characters : string ; links : Optional[ *Substitution[ LinkPack[ HyperLink ] ] ] )

rec type Editor[ HyperLink ] is abstype[ TextPointer ](
     copyText :             proc( Editor[ HyperLink ] );
     cutText :              proc( Editor[ HyperLink ] );
     clearText :            proc();
     pasteText :            proc( Editor[ HyperLink ] );
     insertText :           proc( HyperText[ HyperLink ],bool );
     readFromFile :         proc( file );
     writeToFile :          proc( file );
     select :               proc( TextPointer,TextPointer );
     firstSelection :       proc( -> TextPointer );
     lastSelection :        proc( -> TextPointer );
     firstLine :            proc( -> TextPointer );
     lastLine :             proc( -> TextPointer );
     topLine :              proc( -> TextPointer );
     bottomLine :           proc( -> TextPointer );
     frontOfLine :          proc( TextPointer -> TextPointer );
     endOfLine :            proc( TextPointer -> TextPointer );
     nextLine :             proc( TextPointer -> TextPointer );
     previousLine :         proc( TextPointer -> TextPointer );
     peek :                 proc( -> HyperText[ HyperLink ] );
     read :                 proc( -> HyperText[ HyperLink ] );
     readLine :             proc( -> HyperText[ HyperLink ] );
     selectedText :         proc( -> HyperText[ HyperLink ] );
     before :               proc( TextPointer,TextPointer -> bool );
     endOfText :            proc( -> bool );
     getFont :              proc( -> FontPack );
     getHighlight :         proc( -> bool );
     getProgressIndicator : proc( -> proc( string,real ) );
     getScrollAction :      proc( -> proc( int,bool ) );
     getWindow :            proc( -> Window );
     interactiveEdit :      proc( Editor[ HyperLink ],EventTest,EventTest,EventTest ->
                                  Application );
     invert :               proc( TextPointer,TextPointer );
     lineCount :            proc( -> int );
     new :                  proc();
     offset :               proc( TextPointer,bool -> Index );
     position :             proc( Pos -> TextPointer );
     redisplay :            proc( TextPointer );
     scanLinks :            proc( proc( Substitution[ LinkPack[ HyperLink ] ] -> bool ) );
     scroll :               proc( int,bool );
     search :               proc( HyperText[ HyperLink ],bool -> bool );
     seek :                 proc( Index -> TextPointer );
     setFont :              proc( FontPack );
     setHighlight :         proc( bool );
     setProgressIndicator : proc( proc( string,real ) );
     setScrollAction :      proc( proc( int,bool ) );
     setWindow :            proc( Window );
     unbindWindow :         proc() )
```

6.6 Interface Tools

```
type Appearance is variant( graphical : image ; textual : string )

type EditorPack[ HyperLink ] is
     structure( window :   Window;
                editor :   Editor[ HyperLink ];
                getTitle : proc( -> string );
                getText :  proc( -> HyperText[ HyperLink ] );
                append :   proc( HyperText[ HyperLink ] ) )

type HyperTextPack is EditorPack[ any ]

type MenuEntry is structure( appearance : Appearance ; action : proc( int, MouseEvent ) )

type MenuPack is structure( window :       Window;
                            setTop :       proc( int );
                            getTop :       proc( -> int );
                            setNoVisible : proc( int );
                            getNoVisible : proc( -> int );
                            setHighlight : proc( int,bool );
                            getHighlight : proc( int -> bool );
                            getNoEntries : proc( -> int );

                            entryTable :   Table[ int, MenuEntry ] )

type ButtonPack is structure( window : Window ; flash : proc() )

type SliderPack is structure( window :    Window;
                              set :       proc( real );
                              setBounds : proc( real,real,real ) )

type ChoicePack is structure( window : Window ; set : proc( int,int,bool ) )

type DialoguePack[ HyperLink ] is structure( window : Window ;
                                             set :    proc( HyperText[ HyperLink ] ) )
```

6.7 Programming Environment

```
type Protected is abstype[ i ]( absHolder : structure( abs : i ) )

type ProtectedPack[ T ] is structure( protected :    Protected;
                                      setProtected : any;
                                      getProtected : any;
                                      setConcrete :  proc( T ) ;
                                      getConcrete :  proc( -> T ) )

type WindowState is structure( window : Window ; pos : Pos ; level : Level;
                               open,displayed : bool )

type CompilationError is structure( errorRegion,
                                    errorLine :    CodeRegion;
                                    lineNumber :   int;
                                    errorMessage : string )

type CompilationResult[ TypeDescriptor ] is variant(
     voidResult :     proc();
     nonVoidResult :  proc( -> any );
     typeExpression : TypeDescriptor;
     error :          *CompilationError )

type TypeCompilationResult[ TypeDescriptor ] is variant(
     typeDefinitions : Table[ string, TypeDescriptor ];
     error :          *CompilationError )

type BindingInfo[ Binding ] is structure( binding :    Binding;
```

```
                                        name :       string;
                                        menuOffset : int;
                                        fieldNo :    int )
```

6.8 Concurrency

```
type ThreadPack is abstype[ Thread ]( start :            proc( proc() -> Thread );
                                      getCurrentThread : proc( -> Thread );
                                      getAllThreads :    proc( -> *Thread );
                                      kill,
                                      restart,
                                      suspend :          proc( Thread );
                                      suspendUnlock :    proc( string,Thread ) )

type Semaphore is structure( wait,signal : proc() )
```

6.9 Distribution

```
type RemoteStore is structure( host, storeDir, userName, password : string )

type EnvEntry[ TypeRep ] is structure( entryName : string ; entryType : TypeRep )

type StoreScan[ TypeRep ] is variant( envDescription :  List[ EnvEntry[ TypeRep ] ];
                                      typeDescription : TypeRep )

type RemoteResult[ T ] is variant( successful : T ; error : string )
```

7 Napier88 Releases

7.1 Operating Environment

Napier88 Release 2.0 runs on the following configurations:

- Sun SPARC running SunOs[1] Version 4.1.3.
- Sun SPARC running Solaris Version 1.1.
- DEC Alpha running OSF/1[2] Versions 1.0 or 2.0. This may require modification of the kernel configuration: mail the address given in Section 7.4 for details.

7.2 Obtaining the Napier88 Release

The Napier88 Installation Guide [KBC+94] describes how to obtain a Napier88 release. Napier88 Release 2.0 is Copyright © University of St Andrews 1994 and is subject to a licence fee. If you have a licence for an earlier release of Napier88, however, Release 2.0 is free. The current licence fee is specified in the file:

```
pub/Napier88/release2.0/README
```

which may be obtained by anonymous ftp from the site:

```
ftp-fide.dcs.st-andrews.ac.uk
```

7.3 Napier88 Mailing List

If you wish to be added to an electronic mailing list[3] which carries notifications of future releases and papers, send an e-mail request of the following form:

```
To:         mailbase@mailbase.ac.uk
Subject:

join napier88-users <your-first-name> <your-last-name>
stop
```

where the bracketed words are replaced as appropriate. For example:

```
join napier88-users John Napier
```

The request will be processed automatically and your name added to the mailing list. To send e-mail to all members of the mailing list, send your message to:

```
napier88-users@mailbase.ac.uk
```

---

[1] SunOs™ and Solaris™ are trademarks of Sun Microsystems, Inc.

[2] OSF/1™ is a trademark of the Open Software Foundation.

[3] The mailing list is run by Mailbase™, a service provided by the University of Newcastle upon Tyne.

7.4 Troubleshooting

In the event of problems with downloading a release, or to report any other bugs, send e-mail to:

`napier@dcs.st-andrews.ac.uk`

8 References

[Car88] Cardelli, L. “Building User Interfaces by Direct Manipulation”. In Proc. ACM Symposium on User Interfaces (1988) pp 152-166.

[CDM+90]* Connor, R.C.H., Dearle, A., Morrison, R. & Brown, A.L. “Existentially Quantified Types as a Database Viewing Mechanism”. In **Lecture Notes in Computer Science 416**, Bancilhon, F., Thanos, C. & Tsichritzis, D. (ed), Springer-Verlag, Proc. 2nd International Conference on Extending Database Technology, Venice, Italy (1990) pp 301-315.

[Far91] Farkas, A.M. “ABERDEEN: A Browser allowing intERactive DEclarations and Expressions in Napier88”. University of Adelaide (1991).

[FDK+92]* Farkas, A.M., Dearle, A., Kirby, G.N.C., Cutts, Q.I., Morrison, R. & Connor, R.C.H. “Persistent Program Construction through Browsing and User Gesture with some Typing”. In **Persistent Object Systems**, Albano, A. & Morrison, R. (ed), Springer-Verlag, Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992) pp 376-393.

[KBC+94]* Kirby, G.N.C., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Morrison, R. & Munro, D.S. “The Napier88 Release 2.0 Installation Guide”. University of St Andrews (1994).

[KC93]* Kirby, G.N.C. & Cutts, Q.I. “The Implementation of a Hyper-Programming System”. University of St Andrews Technical Report CS/93/5 (1993).

[KCC+92a] Kirby, G.N.C., Cutts, Q.I., Connor, R.C.H., Dearle, A. & Morrison, R. “Programmers’ Guide to the Napier88 Standard Library, Edition 2.1”. University of St Andrews (1992).

[KCC+92b]* Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. & Morrison, R. “Persistent Hyper-Programs”. In **Persistent Object Systems**, Albano, A. & Morrison, R. (ed), Springer-Verlag, Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992) pp 86-106.

[Kir92]* Kirby, G.N.C. “Reflection and Hyper-Programming in Persistent Programming Systems”. Ph.D. Thesis, University of St Andrews (1992).

[MBB+86] Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A. “A Persistent Graphics Facility for the ICL PERQ Computer”. Software – Practice and Experience 16, 4 (1986) pp 351-367.

[MBC+89a]* Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. “The Napier88 Reference Manual”. Universities of Glasgow and St Andrews Technical Report PPRR-77-89 (1989).

[MBC+89b] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. “Napier88 Release 1.0”. University of St Andrews (1989).

[MBC+94a]* Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C. & Munro, D.S. “The Napier88 Reference Manual (Release 2.0)”. University of St Andrews Technical Report CS/94/8 (1994).

[MBC+94b] Morrison, R., Baker, C., Connor, R.C.H., Cutts, Q.I. & Kirby, G.N.C. “Approaching Integration in Software Environments”. To Appear: Computer Journal (1994).

[MBD+86] Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P. “An Integrated Graphics Programming Environment”. Computer Graphics Forum 5, 2 (1986) pp 147-157.

[Mor82] Morrison, R. “Low Cost Computer Graphics for Micro Computers”. Software – Practice and Experience 12, 8 (1982) pp 767-776.

[SPG91] Silberschatz, A., Peterson, J.L. & Galvin, P.B. **Operating System Concepts**. Addison-Wesley, Reading, Massachusetts (1991).

[Sun89] Sun Microsystems **Open Look™ Graphical User Interface Functional Specification**. Addison-Wesley, Mountain View, California (1989).

Those references marked * are available via *ftp* from the site:

`ftp-fide.dcs.st-andrews.ac.uk/pub/persistence.papers`

or via *World Wide Web* from the URL:

`http://www-fide.dcs.st-andrews.ac.uk:8080/Publications.html`

9 Index