PDF hosted at the Radboud Repository of the Radboud University Nijmegen

The version of the following full text has not yet been defined or was untraceable and may differ from the publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/18841

Please be advised that this information was generated on 2019-01-30 and may be subject to change.
A Tutorial to the CLEAN Object I/O Library

P.M. Achten, M. Wierich

Computing Science Institute/

CSI-R0003 February
A Tutorial to the Clean Object I/O Library - version 1.2

Peter Achten
Martin Wierich
Department of Functional Programming
University of Nijmegen
The Netherlands

February 2, 2000
## Contents

1 Preface .......................................................... 9

2 Introduction ..................................................... 11
   2.1 What are interactive objects ......................... 11
   2.2 How to manage running interactive objects .......... 13
      2.2.1 Opening of interactive objects ................. 14
      2.2.2 Modification of interactive objects .......... 15
      2.2.3 Closing of interactive objects ............... 16
   2.3 How to start an interactive program ................. 16
   2.4 My first Clean object I/O program ................. 17

3 Global structure of the object I/O library ............. 19
   3.1 Abstract devices ...................................... 19
      3.1.1 Menus ........................................... 20
      3.1.2 Windows ........................................ 20
      3.1.3 Timers ......................................... 21
      3.1.4 Receivers ...................................... 21
   3.2 Interactive processes ................................. 22
   3.3 Drawing ................................................. 22
   3.4 Channels ............................................... 22
   3.5 General .................................................... 23

4 Object identification ........................................ 25

5 Drawing .......................................................... 29
   5.1 Pen attributes ........................................ 30
   5.2 Drawing classes ....................................... 31
   5.3 Font and text handling ............................... 32
   5.4 Examples ............................................... 35
      5.4.1 Pen size and position ........................... 36
      5.4.2 Drawing lines ................................... 36
      5.4.3 Drawing text .................................... 37
6 Windows and dialogues
   6.1 Basic terminology .................................. 49
      6.1.1 Anatomy of windows and dialogues .......... 49
      6.1.2 Stacking order ................................ 51
      6.1.3 Active window or dialogue .................. 51
   6.2 Window and dialogue attributes ................. 51
      6.2.1 Window and Dialog attributes ............... 51
      6.2.2 Window attributes ........................... 53
      6.2.3 Dialogue attributes ........................... 54
   6.3 Opening and closing of windows and dialogues ... 54
   6.4 Handling the document layer .................... 55
      6.4.1 Indirect rendering ........................... 55
      6.4.2 Direct rendering ................................ 57
      6.4.3 Pragmatics ................................ 58
      6.4.4 Example: displaying a bitmap ............... 58
   6.5 Handling the control layer ...................... 60
   6.6 Handling the window and dialogue frame ......... 60
      6.6.1 Opening a window or dialogue frame ....... 61
      6.6.2 Changing a window and dialogue frame ........ 61
   6.7 Handling keyboard and mouse input .............. 61
      6.7.1 Keyboard input ................................ 62
      6.7.2 Mouse input ................................ 64
   6.8 Modal dialogues ................................... 66
      6.8.1 Example: a notice extension ................ 67

7 Control handling ...................................... 73
   7.1 The standard controls ........................... 73
      7.1.1 The ButtonControl ........................... 76
      7.1.2 The CheckControl ............................. 77
      7.1.3 The CustomButtonControl ..................... 77
      7.1.4 The CustomControl ............................. 78
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 Timers</td>
<td>117</td>
</tr>
<tr>
<td>9.1 Examples</td>
<td>118</td>
</tr>
<tr>
<td>9.1.1 Expanding circles</td>
<td>118</td>
</tr>
<tr>
<td>9.1.2 Internal clock</td>
<td>121</td>
</tr>
<tr>
<td>10 Receivers</td>
<td>125</td>
</tr>
<tr>
<td>10.1 Receiver definitions</td>
<td>125</td>
</tr>
<tr>
<td>10.2 Receiver creation</td>
<td>126</td>
</tr>
<tr>
<td>10.3 Message passing</td>
<td>127</td>
</tr>
<tr>
<td>10.3.1 Uni-directional message passing</td>
<td>127</td>
</tr>
<tr>
<td>10.3.2 Bi-directional message passing</td>
<td>128</td>
</tr>
<tr>
<td>10.4 Examples</td>
<td>128</td>
</tr>
<tr>
<td>10.4.1 Talk windows</td>
<td>128</td>
</tr>
<tr>
<td>10.4.2 Resetting the counter</td>
<td>132</td>
</tr>
<tr>
<td>10.4.3 Reading the counter</td>
<td>135</td>
</tr>
<tr>
<td>11 Interactive processes</td>
<td>139</td>
</tr>
<tr>
<td>11.1 Defining interactive processes</td>
<td>139</td>
</tr>
<tr>
<td>11.2 Interactive process creation</td>
<td>142</td>
</tr>
<tr>
<td>11.3 Examples</td>
<td>143</td>
</tr>
<tr>
<td>11.3.1 Talk revisited</td>
<td>143</td>
</tr>
<tr>
<td>11.3.2 Clock revisited</td>
<td>146</td>
</tr>
<tr>
<td>12 Clipboard handling</td>
<td>153</td>
</tr>
<tr>
<td>12.1 Example: a clipboard editor</td>
<td>154</td>
</tr>
<tr>
<td>13 Printing</td>
<td>157</td>
</tr>
<tr>
<td>13.1 The user interface for printing</td>
<td>157</td>
</tr>
<tr>
<td>13.2 The print function</td>
<td>158</td>
</tr>
<tr>
<td>13.3 Reusing drawing functions</td>
<td>162</td>
</tr>
<tr>
<td>13.4 The printUpdateFunction function</td>
<td>164</td>
</tr>
<tr>
<td>13.5 The printPagePerPage function</td>
<td>164</td>
</tr>
<tr>
<td>13.6 Printing text</td>
<td>165</td>
</tr>
<tr>
<td>14 TCP</td>
<td>171</td>
</tr>
<tr>
<td>14.1 Introduction to TCP</td>
<td>171</td>
</tr>
<tr>
<td>14.2 Basic ideas</td>
<td>172</td>
</tr>
<tr>
<td>14.3 Blocking TCP in World programs</td>
<td>173</td>
</tr>
<tr>
<td>14.3.1 Establishing a connection</td>
<td>173</td>
</tr>
<tr>
<td>14.3.2 Basic operations on channels</td>
<td>175</td>
</tr>
<tr>
<td>14.3.3 Tearing down a connection</td>
<td>179</td>
</tr>
<tr>
<td>14.3.4 Putting it together</td>
<td>179</td>
</tr>
</tbody>
</table>
14.3.5 Multiplexing ........................................... 180
14.3.6 More channels ........................................... 184
14.4 Non Blocking TCP in GUI Programs ..................... 185

A I/O library .................................................. 191
A.1 StdBitmap .................................................. 191
A.2 StdChannels ............................................... 192
A.3 StdClipboard .............................................. 195
A.4 StdControl ............................................... 196
A.5 StdControlAttribute .................................... 202
A.6 StdControlClass ......................................... 206
A.7 StdControlDef ............................................ 207
A.8 StdControlReceiver ..................................... 209
A.9 StdEventTCP .............................................. 210
A.10 StdFileSelect ............................................ 211
A.11 StdId ..................................................... 212
A.12 StdIO .................................................... 213
A.13 StdIOBasic .............................................. 214
A.14 StdIOCommon ............................................ 216
A.15 StdKey .................................................... 222
A.16 StdMaybe ................................................ 223
A.17 StdMenu .................................................. 224
A.18 StdMenuAttribute ...................................... 227
A.19 StdMenuDef .............................................. 229
A.20 StdMenuElement ......................................... 230
A.21 StdMenuElementClass .................................. 232
A.22 StdMenuReceiver ...................................... 233
A.23 StdPicture ............................................... 234
A.24 StdPictureDef .......................................... 240
A.25 StdPrint ................................................ 242
A.26 StdPrintText ........................................... 245
A.27 StdProcess ............................................... 247
A.28 StdProcessAttribute .................................. 248
A.29 StdProcessDef .......................................... 249
A.30 StdPS t .................................................. 250
A.31 StdPS tClass ............................................ 251
A.32 StdReceiver ............................................. 252
A.33 StdReceiverAttribute ................................ 254
A.34 StdReceiverDef ......................................... 255
A.35 StdSound ................................................ 256
A.36 StdStringChannels ..................................... 257
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.37 StdSystem</td>
<td>258</td>
</tr>
<tr>
<td>A.38 StdTCP</td>
<td>259</td>
</tr>
<tr>
<td>A.39 StdTCPChannels</td>
<td>260</td>
</tr>
<tr>
<td>A.40 StdTCPDef</td>
<td>263</td>
</tr>
<tr>
<td>A.41 StdTime</td>
<td>265</td>
</tr>
<tr>
<td>A.42 StdTimer</td>
<td>266</td>
</tr>
<tr>
<td>A.43 StdTimerAttribute</td>
<td>267</td>
</tr>
<tr>
<td>A.44 StdTimerDef</td>
<td>268</td>
</tr>
<tr>
<td>A.45 StdTimerElementClass</td>
<td>269</td>
</tr>
<tr>
<td>A.46 StdTimerReceiver</td>
<td>270</td>
</tr>
<tr>
<td>A.47 StdWindow</td>
<td>271</td>
</tr>
<tr>
<td>A.48 StdWindowAttribute</td>
<td>276</td>
</tr>
<tr>
<td>A.49 StdWindowDef</td>
<td>278</td>
</tr>
</tbody>
</table>
Chapter 1

Preface

The functional programming language Clean has an extensive library to build graphical user interface applications, the **object I/O library**. In this tutorial the basic concepts of the object I/O library are explained. Many of the concepts are illustrated by means of Clean program examples. Clean code will be typeset in **typewriter** style. The example programs are also available as Clean sources in the corresponding 'Tutorial Examples' folder.

This tutorial is not a technical reference manual; you will not find an extensive and detailed description of every data structure and function of the library. It is also assumed that the reader is familiar with functional programming and Clean.

In Chapter 2 a brief overview of the object I/O library is given, to give the reader a taste of what the object I/O system is all about. Chapter 3 presents the global structure of the object I/O system.

The remaining part of this document explains the individual components of the library. But before we can explain the graphical user interface elements, we first talk about object identification (Chapter 4), and drawing (Chapter 5).

To users of graphical user interfaces the interface elements are of course the **windows** and **dialogues**. These are discussed in Chapter 6. Windows and dialogues can contain **controls**. Because there are many aspects about control handling their treatment deserves a separate chapter (7). In all graphical user interface systems, the set of available commands is presented by means of **menus**, see Chapter 8. To support timing features, **timers** can be used, see Chapter 9. Flexible communication of arbitrary expressions between components can be achieved by using **receivers** and **message passing**, see Chapter 10.

All of the above objects are elements of one interactive process. The object I/O library enables the programmer to split up a large interactive program into several interactive processes that can be created and closed dynamically. This is presented in Chapter 11.

Two final chapters remain that deal with issues that are also related to graphical user interface applications. In Chapter 12 we discuss clipboard handling, a simple user driven mechanism to transfer data between interactive applications. In Chapter 13, we demonstrate the means that exist to send output to a printer.

In Chapter 10 communication of arbitrary Clean expressions between interactive objects is discussed. In Chapter 14 it is demonstrated how Clean programs can communicate with arbitrary programs using **TCP**.

Appendix A contains the definition modules of the Clean object I/O library, version
1.2. in alphabetic order.

The Clean object I/O library is maintained by Peter Achten. Martin Wierich wrote the chapters on printing (Chapter 13) and TCP (Chapter 14) and also developed the corresponding library modules.
Chapter 2

Introduction

In this chapter we give a brief overview of the main features of the object I/O library. We first discuss what the basic components are and how they can be used to construct more complex components (Section 2.1). When these elements have been constructed, they must be opened to create an actual working image on the underlying platform. Elements can be opened and closed dynamically, but it is of course also possible to change them dynamically (Section 2.2). Once we know how to construct graphical user interfaces we can start an interactive program. This is explained in Section 2.3. Finally, to wrap things up Section 2.4 presents the first complete interactive Clean object I/O program of this tutorial, the ubiquitous “Hello world!”.

2.1 What are interactive objects

One way of looking at the object I/O library is to regard it as a collection of building blocks, the interactive objects, that the programmer can use to construct graphical user interfaces. For instance, Figure 2.1 summarises the standard set of control objects that can be placed in a window or dialogue object.

<table>
<thead>
<tr>
<th>Control object:</th>
<th>What does it look like:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ButtonControl</td>
<td><img src="image" alt="Button" /></td>
</tr>
<tr>
<td>CheckControl</td>
<td><img src="image" alt="Check" /></td>
</tr>
<tr>
<td>CompoundControl</td>
<td>Program defined combination of controls</td>
</tr>
<tr>
<td>CustomButtonControl</td>
<td>Program defined button</td>
</tr>
<tr>
<td>CustomControl</td>
<td>Program defined button</td>
</tr>
<tr>
<td>EditControl</td>
<td><img src="image" alt="Just text" /></td>
</tr>
<tr>
<td>LayoutControl</td>
<td>Program defined combination of controls</td>
</tr>
<tr>
<td>PopUpControl</td>
<td><img src="image" alt="PopUpItem" /></td>
</tr>
<tr>
<td>RadioControl</td>
<td><img src="image" alt="Radio" /></td>
</tr>
<tr>
<td>SliderControl</td>
<td><img src="image" alt="Slider" /></td>
</tr>
<tr>
<td>TextControl</td>
<td><img src="image" alt="Text" /></td>
</tr>
</tbody>
</table>

Figure 2.1: The standard set of controls.
All interactive objects are defined by means of algebraic data types. As an example, to define the button control element in the table of Figure 2.1 one would write:

```
button = ButtonControl "Button" []
```

The constituents of this expression are the data constructor `ButtonControl`, applied to the string "Button", and an empty list `[]` of control attributes. This is defined more concisely by the library type definition of a button control element:

```
:: ButtonControl ls pst
    = ButtonControl String [ControlAttribute *(ls,pst)]
```

It should be noted that the names of the type constructor and the data constructor are identical (`ButtonControl`). This convention is used throughout the object I/O library.

The type definition of the button control is parameterised with two type variables: `ls` and `pst`. These correspond to another fundamental characteristic of interactive objects: an interactive object can have local state, and also have an effect on a process state. The type of the local state is identified by the `ls` type parameter, while the type of the process state is identified by the `pst` type parameter. The effect of an interactive object that has a local state of type `ls` and a process state of type `pst` is defined by means of a function of type `(ls,pst) → (ls,pst)`. Such a function is called a callback function. For most interactive objects, the callback function is an attribute of the object. Attributes are also defined by means of algebraic data types. For instance, among many other control attributes, one can find the callback function attribute of controls:

```
:: ControlAttribute st
    = ... | ControlFunction (st->st) | ...
```

Note that the pair of local state and process state constitute the state of an element. The meaning of attributing a control element with a callback function `f` is that when that element is selected by the user, and the current state is the value `(l,p)`, then the new state will be `(f(l,p))`. In other words, a callback function defines a state transition.

Besides having a bag of interactive objects the object I/O library provides programmers glue to construct user interfaces. This glue serves two purposes: (a) from primitive objects one can construct new composite objects, and (b) it puts restrictions on what components are ‘glue compatible’.

The object I/O library has one universal glue `:+:` that can be used to connect two interactive objects that operate on the same local state of type `ls` and process state of type `pst`. Its type definition is as follows:

```
:: :+: t1 t2 ls pst
    = (:+:) infixr 9 (t1 ls pst) (t2 ls pst)
```

In order to define what components are compatible to be glued type constructor classes are applied. The type constructor class `Controls` contains all control elements (see Figure 2.1) but also defines that only `Controls` members can be glued:

```
instance Controls ButtonControl,
    CheckControl,
```
This specification says that \texttt{ButtonControls}, \texttt{CheckControls}, and so on, belong to the same type constructor class, namely \texttt{Controls}. Three of these instances (\texttt{CompoundControl}, \texttt{LayoutControl}, and \texttt{::}) have additional type constructor variables. The only permissible type constructor for these variables should be an element of the \texttt{Controls} class. This is expressed by putting a type constructor context restriction on the variables (the expression followed after \texttt{1}). Let \texttt{button} and \texttt{text} below define a button and a text control respectively:

\begin{verbatim}
button = ButtonControl "Button" []
text = TextControl "Just text" []
\end{verbatim}

then the following expressions are all legal \texttt{Controls} instances: \texttt{button:: button, button:: text, text:: button, and text:: text}.

The collection of interactive objects that is supported by the object I/O library is ordered in four categories, called \textit{abstract devices}.

\textbf{Menus:} Menus provide the set of commands that are available to the user of an interactive program. A program can have an arbitrary number of menus. Menus can be hierarchical, i.e. they can contain menus (\textit{sub menus}) which can contain sub menus as well. Menu items correspond with the menu commands of the program.

\textbf{Receivers:} Receivers are the basic components that interactive objects can use to communicate messages in a flexible way.

\textbf{Timers:} Timers are used by a program to be able to \textit{softly} synchronise actions. A timer basically triggers a callback function every passing of a given time interval.

\textbf{Windows and Dialogues:} Windows provide the primary interface element to the user of a program. Windows can either be \textit{dialogues} or general purpose \textit{windows}. Windows and dialogues can contain arbitrary collections of controls. Analogous to menus, these control collections can be hierarchical, i.e. they can contain collections of controls (\textit{compound controls}), and so on.

\section*{2.2 How to manage running interactive objects}

In the previous section we have had a glimpse of how to define (compositions of) interactive objects. In the object I/O library every interactive object can be \textit{created} and \textit{destroyed} dynamically, but we prefer to call this \textit{opening} and \textit{closing} which sounds more peacefully. Once an interactive object is running the program will need to \textit{modify} it in several ways. Examples are to enable and disable menu elements,
and change the content of a window to reflect the state of the program, and so on. Closing an element is the ultimate modification of a running interactive object. So interactive objects have a life-cycle which consists of three consecutive phases: opening, modification, and closing. Below we discuss these phases.

### 2.2.1 Opening of interactive objects

For each abstract device a function is defined that will open an instance, given a definition. Again type constructor classes are used to control what elements are proper instances of each abstract device. As an example, dialogues are defined by the following type definition:

```haskell
:: Dialog c ls pst = Dialog Title (c ls pst) [WindowAttribute *(ls,pst)]
```

Following the naming convention of the object I/O library, the type variables `ls` and `pst` correspond with the local state of the dialogue and the process state of the program respectively. Callback functions are state transition functions of type \((ls,pst) \rightarrow (ls,pst)\), so the window attribute type constructor is parameterised with \((ls,pst)\). The type constructor class `Dialogs` fixes the instances of dialogues. A `(Dialog c)` is a proper instance of this class, provided that `c` is a proper instance of the `Controls` type constructor class.

```haskell
class Dialogs ddef where
    openDialog :: .ls !(ddef .ls (PSt .l)) !PSt .l
    -> (!ErrorReport, !PSt .l)
    ...

instance Dialogs (Dialog c) | Controls c
```

In Subsection 2.2.2 we will look more closely at the process state argument `PSt`.

The purpose of the `open` function of every abstract device is to map the definition of an interactive object to the ‘physical’ graphical user interface object. It is this ‘physical’ object that can be manipulated by the user of a program (in case of windows, dialogues, and menus) or by the program itself (all abstract devices). The open function returns a value of type `ErrorReport` because it is possible that creating an interactive object fails (lack of system resources or logical inconsistencies). It is good programming practice to check for errors and provide proper feedback to the user.

Let's illustrate the use of the `openDialog` function. Suppose we want to open the following small and not very useful dialogue:

![Hello world dialog box](image)

The program line that does this typically looks something like this:

```haskell
(error, new_process_state)
```
2.2. HOW TO MANAGE RUNNING INTERACTIVE OBJECTS

Once an interactive object has been opened and is in its running phase, it can be modified by the user and the program. For this purpose a running interactive object must be stored somewhere, and it must be identified by the program. Every running interactive object is stored in the I/O state of a program. In Section 2.1 we explained that every interactive object has access to a local state and a process state. The process state is a structured value defined by means of the record type PSt:

```plaintext
:: *PSt 1  // The process state record type
    = { ls :: !1  // The local process state
        , io :: !*IOSt 1  // The I/O state
    }
:: *IOSt 1  // IOSt is an abstract data type
```

A value of abstract type IOSt is created specifically for each interactive process by the object I/O system. In this special value the state of every running interactive object is stored. The other part of a process state, the ls field, can be defined freely by the programmer. In this field, the programmer can store all the data that is required by all interactive objects during the life-cycle of an interactive process. The example programs in this tutorial are usually very small. For such small programs for which no sensible ‘logical’ state has to be defined, the object I/O library provides the convenient type constructor Void:

```plaintext
:: Void = Void
```

The running state of every interactive object is stored in the IOSt. So, a function that modifies the running state of such an interactive object is usually of type (IOSt 1) → (IOSt 1). Some functions such as the abstract device open functions require the PSt record, and consequently have type (PSt 1) → (PSt 1). Because in general the IOSt will contain an arbitrary number of windows, dialogues, menus, timers, and receivers one also has to specify which particular interactive object one intends to modify. For this purpose interactive objects (and their component objects) can be identified by means of Ids. An Id is an abstract type that is generated by the object I/O system. An interactive object is identified by means of a specific Id by adding this Id to the attribute list of the corresponding object definition (see Section 2.1). As an example, for controls the corresponding attribute is:

```plaintext
:: ControlAttribute st
    = ... | ControlId Id | ...
```

The major part of the object I/O library defines the modification functions that the programmer can use to modify a running interactive object.

As an example, suppose we want to change the content of the text control of the “Hello world!” example to “Goodbye world!”. To do this, we need to identify the
text control. A control is identified uniquely by its personal Id. So we add a ControlId attribute to the attribute list of the text control. If textid identifies the text control then the corresponding program fragments look something like:

\[
\begin{align*}
\text{(error,new\_process\_state)} \\
= \text{openDialog} \\
\quad \text{my\_local\_state} \\
\quad \text{(Dialog "")} \\
\quad \text{(TextControl "Hello world!" [ControlId textid])} \\
\quad \text{[]} \\
\quad \text{the\_process\_state}
\end{align*}
\]

\[
\begin{align*}
\text{changeText\_process\_state} & =: \{\text{io}\} \\
& = \{\text{process\_state \& io=\text{setControlText textid "Goodbye world!" io}}\}
\end{align*}
\]

2.2.3 Closing of interactive objects

As explained in the previous two subsections, once an interactive object has been opened it is in its running phase and it will remain in that state until it is explicitly closed by the program. Note that although it may seem to the user that he is able to close a window, it is actually the program that responds to a user request to really close a window. For all abstract devices and their components, there are close operations. A close operation will remove the ‘physical’ graphical user interface element and free system resources that were required to operate the interactive object properly. In addition, it is also removed from the IOSt. Closing an interactive object also closes its component objects, so closing a dialogue automatically closes all controls that are part of the dialogue.

2.3 How to start an interactive program

The starting point of every Clean program (interactive or not) is the Start function. The essence of an interactive program is that it is a function that can change the world. So for interactive programs the Start function must have type *World \to *World. The typical appearance of the Start function of an interactive program looks something like:

\[
\begin{align*}
\text{Start} & :: \ast \text{World} \to \ast \text{World} \\
\text{Start \ text} & = \ldots \ \text{world}
\end{align*}
\]

In the previous sections we have seen how to define interactive objects and get them running. The abstract device open functions require a PSt value, containing an IOSt value. These are created by the object I/O system using the function startIO.

\[
\begin{align*}
\text{startIO} & :: \!\text{Document\_Interface} \!\text{.1 } (\text{Process\_Init (PSt .1))} \\
& \quad \!\text{[Process\_Attribute (PSt .1)]} \\
& \quad \!\ast \text{World} \to \ast \text{World}
\end{align*}
\]

:: Process\_Init pst ::= \text{IdFun pst} \\
:: \text{IdFun} \quad \text{st} ::= \text{st} \to \text{st}
The details of this function will be explained in Chapter 11. Briefly, it creates an initial process state given the initial local process state of type \textit{l}. This initial process state will contain a tailor-made \textit{IOSt} value. In this way one switches from the world environment to the process state environment, and the interactive program can be initialised. This is done by the initialisation function (of type \texttt{(ProcessInit (PSt l)})). In this function the programmer can create the initial interactive objects that are required by the program.

After initialisation the interactive process will be evaluated. Evaluation means that the object I/O system takes care that the proper process state transition functions will be applied that are specified by the program. For instance, when the user presses a button or enters text in some text edit field, this generates low-level events. These events are interpreted by the object I/O system and mapped to abstract events such as “this button was pressed” and “this keyboard input was entered”. If the interactive object has a callback function (given as an attribute) then this function is applied to its current local state and process state. This returns a new local state and process state, with which the object I/O system continues.

The evaluation of an interactive process terminates as soon as it becomes \textit{closed}. The only way for an interactive process to be closed is by means of the process modification function \texttt{closeProcess}:

\begin{verbatim}
closeProcess :: !(PSt .1) -> PSt .1
\end{verbatim}

This function closes all currently running interactive objects of the interactive process and returns a process state that contains an empty \textit{IOSt} value. All modification operations have no effect when applied to an empty \textit{IOSt} value. If an interactive program consists of one interactive process, then the application of \texttt{closeProcess} suffices to terminate the evaluation of \texttt{startIO}. In Chapter 11 it is discussed how an interactive program can open and close interactive processes dynamically. Note that if an interactive program does not close all of its processes, it will run on forever.

2.4 My first Clean object I/O program

To complete the introduction we present the Clean object I/O version of the well known “Hello world!” program. Here it is:

\begin{verbatim}
module hello

// ******************************************************************************
// Clean tutorial example program.
//
// This program creates a dialog that displays the "Hello world!" text.
// ******************************************************************************

import StdEnv, StdIO

Start :: *World -> *World
Start world
    = startIO NDI Void initialise [] world
where

initialise pst
    # (error pst)   = openDialog Void hello pst
    | error <> NoError = closeProcess pst
    | otherwise       = pst

    hello = Dialog ""
\end{verbatim}
This program creates an interactive process which opens the same dialogue as shown earlier in Section 2.2.1. The singleton type Void introduced earlier is used to specify that this program has no interesting local process state (the second argument of startIO), and also no interesting local dialogue state (the first argument of openDialog).

The dialogue has one attribute, the callback function that should be applied in case the user wants to close the dialogue. In this case closing the dialogue will close the “Hello world!” program. So the callback function can simply be closeProcess. However, the type of a callback function of an interactive object also operates on a local state (which is Void in this case). To conveniently transform a function of type $(a \rightarrow b)$ into a function of type $(c,a) \rightarrow (c,b)$, the library function noLS :: $(a \rightarrow b) \rightarrow (c,a) \rightarrow (c,b)$ is applied.
Chapter 3

Global structure of the object I/O library

The application programmer’s interface of the Clean object I/O library currently consists of 49 definition modules (they are all given in Appendix A). These modules contain everything you need to create interactive Clean programs with. No other modules and no other symbols should be imported from the object I/O library. Violation of this rule can result in error-prone applications at worst and non-portability at least. In this chapter the module structure of the API is discussed. This will help you to find your way quickly in the object I/O library.

As a global naming convention, all definition modules have the prefix Std. The module StdIO is a convenience module that collects all modules that you normally need for graphical user interface programs. The module StdTCP is a convenience module that collects all modules required for TCP programming. The modules can be divided roughly into five major categories: the abstract devices, interactive processes, drawing, channels and general.

3.1 Abstract devices

Abstract devices have been introduced in Section 2.1 (page 13). These are the menu, window, timer, and receiver device. These devices occupy most of the modules and type definitions of the object I/O library. The following naming conventions have been employed for these modules:

- The names of the modules that contain type definitions to define abstract device instances have postfix Def. So menu definitions can be found in the module StdMenuDef, receiver definitions can be found in the module StdReceiverDef, and so on. Although controls are not an abstract device, a definition module also exists for controls, namely StdControlDef.

- For each module that contains type definitions to define abstract device instances (so of the form StdObjectDef) there also exists a module that defines basic access functions on its attributes. The corresponding name is formed as StdObjectAttribute. So, control attribute access functions can be found in the module StdControlAttribute.

- Abstract device instances that consist of elements use type constructor classes to enumerate their elements. The corresponding type constructor classes and
standard instances are defined in the modules which names end with \textit{Class}. So \textit{menu elements} can be found in the module \texttt{StdMenuElementClass}. The \textit{controls} can be found in \texttt{StdControlClass}.

- Receivers are non standard elements of some abstract device instances. Given the name \textit{Object} of the parent device instance, you can find the type constructor class instance declarations in the modules which names are formed like \texttt{StdObjectReceiver}. So, receiver instances of timers can be found in \texttt{StdTimerReceiver}.

- The operations on an object \textit{Object} can be found in the module named \texttt{StdObject}. So \textit{window} operations can be found in the module \texttt{StdWindow}, \textit{menu element} operations can be found in the module \texttt{StdMenuElement}, and so on.

Below we discuss the module structure of each of the abstract devices.

3.1.1 Menus

The API for menus and menu elements consists of six modules:

\texttt{StdMenuDef} \\
contains the type definitions of menus, pop up menus, and menu elements.

\texttt{StdMenuAttribute} \\
defines the access functions on menu (element) attributes.

\texttt{StdMenuElementClass} \\
defines the menu element instances for menus and pop up menus. These have different type constructor classes (\texttt{MenuElements} and \texttt{PopUpMenuElements} respectively) because pop up menus are not allowed to contain sub menus.

\texttt{StdMenuReceiver} \\
adds \textit{receivers} as a proper instance of menu elements.

\texttt{StdMenuElement} \\
contains all operations on menu elements.

\texttt{StdMenu} \\
contains all operations on menus. Here also the \texttt{Menus} type constructor class is given together with its two instances \texttt{Menu} and \texttt{PopUpMenu}.

3.1.2 Windows

The API for windows, dialogues, and controls consists of eight modules:

\texttt{StdWindowDef} \\
contains the type definitions of windows and dialogues.

\texttt{StdControlDef} \\
contains the type definitions of controls. These can be placed in windows and dialogues.

\texttt{StdWindowAttribute} \\
defines basic access functions on window attributes.
3.1. ABSTRACT DEVICES

StdControlAttribute
defines basic access functions on control attributes.

StdControlClass
defines the control instances for windows and dialogues. For windows and
dialogues these belong to the Controls type constructor class.

StdControlReceiver
adds receivers as a proper instance of controls.

StdControl
contains all operations on controls.

StdWindow
contains all operations on windows and dialogues. Here also the Windows
and Dialogs type constructor classes are given together with their instances
Window and Dialog.

3.1.3 Timers

The API for timers consists of five modules:

StdTimerDef
contains the type definitions of timers and its elements.

StdTimerAttribute
defines basic access functions on timer attributes.

StdTimerElementClass
defines the timer elements instances. These are currently only glueing in-
stances.

StdTimerReceiver
adds receivers as a proper instance of timer elements. This is actually the
most interesting timer element.

StdTimer
contains all operations on timers.

3.1.4 Receivers

The API for receivers consists of three modules:

StdReceiverDef
contains the type definitions for uni-directional and bi-directional receivers.

StdReceiverAttribute
defines basic access functions on receiver attributes.

StdReceiver
contains all operations on receivers. Here you can also find the message passing
functions.
3.2 Interactive processes

As stated in Section 2.3, every interactive program has to be opened as an interactive process. The API for interactive processes consists of three modules:

**StdProcessDef**
- contains the type definitions for interactive processes.

**StdProcessAttribute**
- defines basic access functions on process attributes.

**StdProcess**
- contains all operations on processes including the functions mentioned in Chapter 2, `startIO` and `closeProcess`. The `startIO` function is actually a specialised version of the more general type constructor class `Processes` which allows you to create a topology of interactive processes.

3.3 Drawing

In graphical user interface applications, graphics play an important role. Virtually every interface object has a visual representation that is drawn by the underlying platform. Drawing operations will be required by most applications to give the user visual feedback on the current documents that are being manipulated or the status of controls. The manipulation of text is also an issue in drawing information. Text can be presented in very different fonts, sizes, and variations. Drawing on screen and printing on paper is very analogous. With some care rendering can be abstracted from the output device.

The API for drawing consists of five modules:

**StdPictureDef**
- contains the type definitions of drawable elements, fonts, colours, pen attributes, and some platform dependent constants.

**StdPicture**
- contains all drawing operations on pictures. These are structured by means of the type constructor classes `Drawables` for drawing figures, and `Fillables` for filling figures. Picture attributes can be retrieved and modified.

**StdBitmap**
- adds *bitmaps* as a proper instance of the `Drawables` type constructor class. Currently the set of bitmap operations is very limited.

**StdPrint**
- defines very general printing functions. Printing is the same as drawing, except that the output is sent to a printer rather than some screen object.

**StdPrintText**
- builds on `StdPrint` and offers some specialised functions to print text.

3.4 Channels

Internally, interactive processes that belong to the same interactive program can communicate arbitrary functional values by means of receivers (see Section 3.1.4).
To allow (interactive) Clean programs to communicate with other applications an extensive TCP/IP API is provided. *Channels* are the medium for communication.

The API for using channels consists of five modules:

**StdChannels**
contains operations to send and receive on channels.

**StdTCPDef**
contains various type definitions needed for event driven communication.

**StdTCPChannels**
defines instances for most of the classes defined in *StdChannels*.

**StdStringChannels**
defines instances to send and receive strings.

**StdEventTCP**
contains functions for using event driven TCP.

### 3.5 General

Finally, there are a number of modules that are less easily categorised. These are the following eleven modules:

**StdClipboard**
In this module *clipboard* operations are defined. Clipboard operations are defined in more detail in Chapter 12.

**StdFileSelect**
In this module two functions are defined by which a user can open platform dependent directory browsing dialogues.

**StdId**
In this module the *identification* value generating functions are defined. This is discussed in more detail in Chapter 4.

**StdIOBasic, StdIOCommon** and **StdKey**
These modules provide a lot of type definitions and functions that are needed by many of the abstract device modules introduced in Section 3.1.

**StdMaybe**
In this module a type is introduced that is very useful for providing optional results and optional arguments. It is used by many operations in the abstract device modules.

**StdPSt** and **StdPStClass**
In these modules operations are collected on the process state that are not related to any abstract device. *StdPSt* contains several type constructor classes for file and time access. Their instances can be found in *StdPStClass*. Other frequently used functions in *StdPSt* are the 'lifting' functions defined on the process state. With these lifting functions one can easily transform for instance an *IOSt* transition function to a *PSt* transition function.

**StdSound**
This is an experimental module introduced to provide access to sound files.
StdTime

In this module some time access operations can be found that can be used independently of the timer device.
Chapter 4

Object identification

Before we can really delve into the details of the object I/O library we first need to learn how to identify running interactive objects. As we have briefly discussed in Section 2.2.2, all objects can have an *identification attribute*. An identification attribute is a value of type Id. Because attributes are optional, the programmer is not forced to provide a value for this attribute. Objects without identification attribute can not be modified at run-time. If the program needs to modify an interactive object, it must have been provided with an identification attribute.

The type Id is an abstract data type, and you can import it via the module StdId. *All Ids* are generated by the system. The type constructor class Ids defines the creation functions. Ids can be created from the World, IOSt, and the PSt environment. Every new Id taken from these environments is guaranteed to be fresh with respect to the other Ids generated by any of these functions.

```haskell
:: Id

class Ids env where
  openId :: !*env -> (! Id, !*env)
  openIds :: !Int !*env -> (!![Id], !*env)

  openRId :: !*env -> (! RId m, !*env)
  openRIds :: !Int !*env -> (!![RId m], !*env)

  openR2Id :: !*env -> (! R2Id m r, !*env)
  openR2Ids :: !Int !*env -> (!![R2Id m r], !*env)

instance Ids World,
             IOSt .1,
             PSt .1
```

As the Ids class definition shows, Id values are not the only identification values that are used in the object I/O system. Two special kinds of identification values of type (RId m) and (R2Id m r) are used to identify receivers that accept messages of type m in the first case, and in addition reply with a message of type r in the second case. Receivers are discussed in Chapter 10.

The purpose of having Ids is to unambiguously identify running interactive objects. When assigning Ids to interactive objects, a program must comply to the Id *assignment rule*: at all times during the life-time of an interactive program, an
CHAPTER 4. OBJECT IDENTIFICATION

identification value (Id, RId m, or R2Id m r) must be bound to at most one running interactive object.

The abstract device open functions check whether the interactive object definition argument is valid with respect to the Id assignment rule. If this is not the case, the ErrorReport alternative ErrorIdsInUse is returned, and the interactive object will not be opened. If the Id assignment rule is not violated, and no other error occurred, then the alternative NoError is returned.

For simple programs such as the “Hello world!” program in Section 2.4 no identification values have to be created. For somewhat longer programs that fit easily inside one module (such as most examples in this tutorial) it is usually most convenient to create the required identification values in the Start function and pass these to the initialisation function argument of startIO (Section 2.3). In the left-hand side of the initialisation function one can introduce names to these identification values that can then be used in all locally defined callback functions. So the outline of such a program is as follows:

```
module name_of_program
import StdEnv, StdIO  // Import of standard libraries

Start :: *World -> *World
Start world
  # (ids,world) = openIds n world  // Create n Ids
  - startIO SDI Void (initialise ids) [] world  // Pass these to initialize
initialise :: [Id] (PSt .1) -> PSt .1
initialise [id1,id2,...,idn] pst  // Introduce names for all n Ids
  - ...  // Create interactive objects
where

  ...  // Local callback function
  - ... id1 ... id2 ...  // can now use the Ids
```

The advantage of this scheme is that if good names are given to the Id values it is easier to make no mistakes.

Another scheme that is more useful in case it makes less sense to define all callback functions locally to the initialisation function is to put the Id values in a record in the local process state. Here, the record field names take care of naming the Ids. One can define a record MyIds that contains all Id entries (possibly hierarchically). If program states become more involved it is also a good idea to define them as a record because this makes it easier to change the program in the future. So, one can also introduce some local process state record MyState containing MyIds. Although not strictly necessary, it is also convenient to define a function that creates a MyIds record from any environment that belongs to the Ids type constructor class. Such a function has the overloaded type:

```
openMyIds :: !*env -> (MyIds,!*env) | Ids env
```

This gives you more flexibility where to actually create the MyIds record because this function can be applied to any environment instance of the Ids type constructor class. The outline of a program using this scheme is as follows:

```
module name_of_program
import StdEnv, StdIO  // Import of standard libraries

:: MyIds  // The record that stores Ids
- { id1 :: Id
  , id2 :: Id
  , ...
  , idn :: Id
  }
:: MyState // The local process state
- { myIds :: MyIds // contains the MyIds record
  , ...
  }

openMyIds :: !*env -> (!MyIds,!*env) | Ids env // openMyIds creates the Ids
openMyIds env
  # ((id1,id2,...idn).world) = openIds n world // Create n Ids
  = ((id1-id1,id2-id2,...,idn-idn).env) // Create MyIds record

Start :: *World -> *World
Start world
  # (myIds,world) = openMyIds world // Create MyIds record
  # myState = {...myIds=myIds...} // in the local process state
  = startIO SDI myState initialise [] world

callback :: (LocalState,PSt MyState) -> (LocalState,PSt MyState)
callback (ls,pst=:{ls=(myIds)}) // Every callback function
  = ... myIds.id1 ... myIds.id2 ... // can now use the Ids
Chapter 5

Drawing

In a graphical user interface most of the information that is presented to users is drawn, such as the shape of windows, dialogues, controls, but also handling of text. This central issue is handled in this chapter. All drawing functions require an environment of type *Picture. A Picture is created for each interactive object that can be drawn in. The life-cycle of a Picture environment is equal to the life-cycle of its parent object. Also for printing a Picture environment is used (printing is discussed in detail in Chapter 13).

A Picture defines a coordinate system for drawing operations (see figure 5.1).

![Figure 5.1: Picture coordinates.](image)

X-axis coordinates increase from left to right, Y-axis coordinates increase from top to bottom. The range for both axes is defined by the macro `viewDomainRange` that can be found in module `StdIOCommon`:

```plaintext
viewDomainRange ::= { corner1 = \{x = 0-(2^30),y = 0-(2^30)\} \
                    , corner2 = \{x = 2^30,y = 2^30 \} 
                    }
```

Drawing operations on a Picture use the coordinate system to define where objects
should be drawn. The objects themselves are made up of the pixels, lying between the coordinates. Figure 5.2 zooms in on the coordinate system from zero and increasing. The pixels on x-coordinates 0, 5, 10,… and y-coordinates 0, 5, 10,… are displayed.

![Figure 5.2: Picture coordinates and pixels](image)

### 5.1 Pen attributes

Drawing a figure is done using a *pen*. The pen determines which pixels should be drawn and in what colour. Like most other elements in the object I/O library, pens have attributes. These are the following (they can be found in module `StdPictureDef`):

```haskell
:: PenAttribute  // Default:
= PenSize  Int  // 1
| PenPos  Point2  // zero
| PenColour  Colour  // Black
| PenBack  Colour  // White
| PenFont  Font  // DefaultFont
```

**PenSize**

This attribute defines the width and height of the drawing pen. The default value is 1, which means that drawing a point will fill an area of 1 pixel wide and 1 pixel high. Negative or zero values are always set to 1.

**PenPos**

This attribute determines the current position of the drawing pen. Its default value is `zero`. The type definition of `Point2` is:

```haskell
:: Point2 = {x::Int,y::Int}
```

instance zero Point2 where

zero = {x=0,y=0}
PenColour

This attribute determines the colour drawn pixels will have. The default value is black. The Colour data type is defined in module StdPictureDef:

```haskell
class Colour
    = Black | DarkGrey | Grey | LightGrey | White
    | Red | Green | Blue
    | Cyan | Magenta | Yellow
    | RGB RGBColour
```

A colour can range between black and white (first five alternatives defining 100%, 75%, 50%, 25%, and 0% blackness), be one of red, green, blue, be one of cyan, magenta, yellow, or some custom defined combination of red, green, blue components. Currently the library does not support colour tables or palette management operations, so the use of RGB colours tends to be speculative.

PenBack

This attribute determines the colour erased pixels will have. The default colour is white.

PenFont

This attribute sets the current font that will be used when drawing text. Drawing text is not affected by the current width and height of the pen.

The current individual pen attributes can be set and retrieved by corresponding functions setPenAttribute and getPenAttribute respectively. It is also possible to set several pen attributes at once using setPenAttributes and retrieve all current pen attributes with getPenAttributes. These functions can all be found in StdPicture.

5.2 Drawing classes

The drawing operations are divided into three groups, ordered by means of type constructor classes:

**Drawables**

draws (draw and drawAt) or erases (undraw and undrawAt) its instances. These are characters, strings, vectors, ovals, curves, boxes, rectangles, polygons, and bitmaps.

```haskell
class Drawables figure where
    draw :: figure !*Picture -> *Picture
    drawAt :: !Point2 !*Picture -> *Picture
    undraw :: figure !*Picture -> *Picture
    undrawAt :: !Point2 !*Picture -> *Picture
```

**Fillables**

fills (fill and fillAt) or erases (unfill and unfillAt) its instances. These are ovals, curves, boxes, rectangles, and polygons.

```haskell
class Fillables figure where
    fill :: figure !*Picture -> *Picture
    fillAt :: !Point2 !*Picture -> *Picture
    unfill :: figure !*Picture -> *Picture
    unfillAt :: !Point2 !*Picture -> *Picture
```
class Fillables figure where
    fill :: !figure !Picture -> *Picture
    fillAt :: !Point2 !figure !Picture -> *Picture
    unfill :: !figure !Picture -> *Picture
    unfillAt :: !Point2 !figure !Picture -> *Picture

Hilites
fills the interior of its instances in such a way that the current picture content remains visible. Its instances are boxes and rectangles.

class Hilites figure where
    hilite :: !figure !Picture -> *Picture
    hiliteAt :: !Point2 !figure !Picture -> *Picture

Each of these type constructor classes allows its elements to be drawn at the current pen position or at an absolute pen position. Because of this reason the data type definition of most of these elements do not specify their location. Exceptions are rectangles, lines, and points.

5.3 Font and text handling

When working with text you frequently will want to know the dimensions of the text for layout purposes or simply to calculate the size of an element containing that text. The dimensions of a piece of text depends on two parameters:

- The font is an abstract value that describes the shape of a text. The usual way to identify a font is by its name, point size, and style variations.
- The drawing environment (Picture) determines the actual size in terms of a resolution dependent unit. The resolution of the screen is usually a lot smaller than the resolution of a laser writer.

Because there is a great variance of available fonts and drawing environments per machine writing a program that handles fonts properly requires some care. Font operations are ordered in three groups.

The first group of font operations return information about the currently available fonts. The function getFontNames returns a list of the names of all available fonts. Given an element of this list, the functions getFontStyles and getFontSizes return for that particular font the available style variations and sizes. Because in modern font management systems for many fonts no restriction exists on the size, the function getFontSizes is also parameterised with two bounding size arguments. Their type definitions are:

g.getFontNames :: ![Picture] -> ![FontName] ![Picture]
g.getFontStyles :: ![FontName] ![Picture] -> ![FontStyle] ![Picture]
g.getFontSizes :: ![Int] ![Int] ![FontName] ![Picture] -> ![FontSize] ![Picture]

The second group of font operations opens fonts. The function openFont creates a value of type Font given a font definition. A font definition is a record of type FontDef and is defined as follows:
:: FontDef
    = { fName :: !FontName
        , fStyles :: ![FontStyle]
        , fSize :: !FontSize
    }

Because there are so many different font systems and style variations both font names and style variations are of type String. If you want to be sure that you are selecting an existing font use the functions of the first group. In any case, if the font definition argument of openFont does not correspond with an installed font, then it returns a False Boolean and a font value that is supposed to be the closest matching font available. If it succeeds to find a matching font it will return a True Boolean and that font. The type of openFont is:

openFont :: !FontDef !*Picture -> (!(!Bool,!Font),!*Picture)

There are two functions that open the font that is used by default in a document window (openDefaultFont) and the font that is used by the system for dialogs, controls, window titles and so on (openDialogFont). These fonts are of course always available. Their types are:

openDefaultFont :: !*Picture -> (!Font,!*Picture)
openDialogFont :: !*Picture -> (!Font,!*Picture)

The last group of font operations is related to font and text metrics. Font metrics are retrieved by the functions getPenFontMetrics and getFontMetrics. The first retrieves the metrics of the current pen Font, while the second retrieves the metrics of the argument Font.

getPenFontMetrics :: !*Picture -> (!FontMetrics,!*Picture)
getFontMetrics :: !Font !*Picture -> (!FontMetrics,!*Picture)

The metrics of a font consists of three height related values (leading, ascent, and descent), illustrated in Figure 5.3 and one width related value.

The width related value (max. width) is the maximum width of all characters in that particular font. For non-proportional fonts such as Courier this implies that the width of all characters is identical, so “iii” is just as wide as “mmm”. For proportional fonts such as this text the width of characters can vary a lot. Compare for instance the width of the text “iii” with “mmm”. The metrics of a font are collected in a record of type FontMetrics:

:: FontMetrics
    = { fAscent :: !Int  // The ascent of the font
        , fDescent :: !Int  // The descent of the font
        , fLeading :: !Int  // The leading of the font
        , fMaxWidth:: !Int  // The max. width of the font
    }

The final functions in the last group of font operations are used for calculating the width of a (list of) character(s), and a (list of) string(s). As with the font metrics functions, these functions also come in pairs, one using the current pen Font, and the other using the argument Font.
There is a subtle difference in calculating the width of one character versus the width of one string. The width of a character is determined by the character only. The width of a string can depend on the order of the characters it contains. A font system can take advantage of the fact that some adjacent characters can be placed more closely together to obtain a better looking result when drawing the string. This is called kerning. In the object I/O system, the programmer can rely on the fact that if a piece of text is drawn character by character then the character width function returns the correct width of the drawn character. If a piece of text is drawn by using a string, then the string width function returns the correct width of the drawn string.

Because there is wide variety of fonts available the StdPictureDef module provides a number of macros that help you make a program less dependent on the set of available fonts. The following macros provide a number of font definitions that are guaranteed to be available on the platform:

<table>
<thead>
<tr>
<th>Font macros</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SerifFontDef</td>
<td>Garamond, Times</td>
</tr>
<tr>
<td>SansSerifFontDef</td>
<td>Helvetica</td>
</tr>
<tr>
<td>SmallFontDef</td>
<td>“This is a small text”</td>
</tr>
<tr>
<td>NonProportionalFontDef</td>
<td>Courier</td>
</tr>
<tr>
<td>SymbolFontDef</td>
<td>$\forall \exists \alpha \beta \leftrightarrow$</td>
</tr>
</tbody>
</table>
5.4 EXAMPLES

The following macros provide a number of standard font variations that are guaranteed to be available on the platform:

<table>
<thead>
<tr>
<th>Style macros:</th>
<th>Example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ItalicStyle</td>
<td>Madam, I'm Adam</td>
</tr>
<tr>
<td>BoldStyle</td>
<td>Madam, I'm Adam</td>
</tr>
<tr>
<td>UnderlinedStyle</td>
<td>Madam, I'm Adam</td>
</tr>
</tbody>
</table>

5.4 Examples

In this section we give small examples of all of the drawable elements. Each of the examples is defined by a drawing function `example` of type `*Picture -> *Picture`. To actually get something on the screen one can use the following drawing framework program `drawingframe`. Figure 5.4 gives a snapshot of the framework program.

![Drawingframe window](image)

Figure 5.4: The framework window.

```haskell
module drawingframe

// ******************************************************************************
// Clean tutorial example program.
// //
// // This program defines a framework in which one can test drawing functions.
// // The program relies on a function, example, of type *Picture -> *Picture.
// // ******************************************************************************

import StdEnv, StdIO

Start :: *World -> *World
Start world = startIO SDI
  Void
  (snd o openWindow Void testwindow)
  [ProcessClose closeProcess]
  world

  testwindow = Window "Test Drawing" NilLS
    [ WindowViewSize size
      , WindowClose (noLS closeProcess)
      , WindowLook True (_, _->example)
      , WindowViewDomain {corner1=origin,corner2=maxdomain}
    ]

    where
        size     = {w=200,h=60}
        origin   = {x=(-20),y=(-20)}
        maxdomain = {x=origin.x+size.w,y=origin.y+size.h}

    // Here, example draws the string "Pop" at zero
    example = drawAt zero "Pop"
```
5.4.1 Pen size and position

Given a pen position \( \{x, y\} \), drawing a point, line, always occurs to the right and below the pen position. Figure 5.5 illustrates these cases: from left to right the following example functions are applied respectively:

\[
\text{example} = \text{drawPointAt zero } \circ (\text{setPenSize } 1) \\
\text{example} = \text{drawPointAt zero } \circ (\text{setPenSize } 2) \\
\text{example} = \text{drawPointAt zero } \circ (\text{setPenSize } 3)
\]

![Figure 5.5: Drawing a point at zero with different pen sizes.](image)

5.4.2 Drawing lines

The shape of a line is influenced by the PenSize attribute in the same way as the shape of points. Figure 5.6 shows the result of drawing lines with pen sizes 1, 2, and 3 respectively:

\[
\text{example} = \text{drawLine zero } \{x=5, y=5\} \circ (\text{setPenSize } 1) \\
\text{example} = \text{drawLine zero } \{x=5, y=5\} \circ (\text{setPenSize } 2) \\
\text{example} = \text{drawLine zero } \{x=5, y=5\} \circ (\text{setPenSize } 3)
\]

![Figure 5.6: Drawing a line from zero to \{x=5, y=5\} with different pen sizes.](image)

There are several ways to draw lines. The function \texttt{drawLineTo} draws a line from the current pen position to the argument point. If these happen to be equal, then the result is the same as \texttt{drawPointAt} with the same argument. The new pen position is the same as the target point. The function \texttt{drawLine} draws a line from the first argument point to the second argument point without changing the pen position.

Lines can also be drawn using the Vector2 instance functions from the Drawable type constructor class. The function \texttt{draw} applied to a vector \( \{vx, vy\} \) draws a line
from the current pen position \(\{x,y\}\) to the point \(\{x=x+v_x, y=y+v_y\}\). The function `drawAt` applied to a point \(\{x,y\}\) and a vector \(\{v_x,v_y\}\) draws a line from \(\{x,y\}\) to the point \(\{x=x+v_x, y=y+v_y\}\).

### 5.4.3 Drawing text

Given a pen position \(\{x,y\}\), drawing a piece of text (Char or String) will always draw the text using the current PenFont. The shape of the drawn characters relies only on the font information, not on the current PenSize. Text can be drawn in any of the available colours. The baseline of the particular font determines the position of the first character, which is drawn with its left baseline starting at \(\{x,y\}\). Figure 5.7 shows the result of the following `example` function:

\[
\text{example} = \text{drawAt} \text{ zero} \text{ "Pop"}
\]

![Figure 5.7: Drawing the text "Pop" at zero.](image)

The new pen position is, in this case, \(\{x=24, y=0\}\). One might expect the new pen position to be \(\{x=22, y=0\}\), but usually the horizontal character space is also included. This facilitates drawing text character by character. But some caution should be taken. One might expect that the function

\[
\text{example} = \text{draw} \text{ "p"} \circ \text{draw} \text{ "o"} \circ \text{draw} \text{ "P"}
\]

produces the same result, but this depends on the font (as explained in Section 5.3), so in general one should not assume that this is the case. The only certain way to know how much the pen position will change in case of text is by calculating the width of the same text, or by comparing the pen positions before and after drawing.

### 5.4.4 Drawing ovals

An Oval is a transformed unit circle defined by a horizontal radius, `oval_rx` and a vertical radius, `oval_ry`. For each point \(\{x,y\}\) on a unit circle, its corresponding point on the oval is given by \(\{x=x*\text{oval}_r, y=y*\text{oval}_r\}\). The type definition of an Oval is:

\[
:: \text{Oval} = \{\text{oval}_r::\text{Int}, \text{oval}_r::\text{Int}\}
\]

Both radius values are always taken to be at least zero. If any of these values is negative, then zero is used instead. Ovals are drawn using the `Oval` instance functions.
from the Drawables type constructor class. The function \texttt{draw} uses the current pen position as the center of the oval. The function \texttt{drawAt} uses the argument Points2 as the center of the oval. Drawing an oval does not change the pen position. In case one of the radius values is taken to be zero drawing the oval displays nothing. Figure 5.8 shows the result of drawing three ovals at zero defined as follows:

\begin{verbatim}
example = drawAt zero {oval_rx=5, oval_ry=3}
example = drawAt zero {oval_rx=5, oval_ry=5}
example = drawAt zero {oval_rx=3, oval_ry=5}
\end{verbatim}

![Figure 5.8: Three oval shapes drawn at zero.](image)

The shape of an oval is also affected by the current \texttt{PenSize} attribute. Increasing the pen size does not increase the outline of the oval. The only pixels that are affected are inside the oval. Figure 5.9 shows the center oval of Figure 5.8 when drawn with pen size of 1, 2, and 3.

\begin{verbatim}
example = drawAt zero {oval_rx=5, oval_ry=5} o (setPenSize 1)
example = drawAt zero {oval_rx=5, oval_ry=5} o (setPenSize 2)
example = drawAt zero {oval_rx=5, oval_ry=5} o (setPenSize 3)
\end{verbatim}

![Figure 5.9: Three oval shapes drawn with increasing pen sizes.](image)

Ovals are also an instance of the Fillables type constructor class. The function \texttt{fill} and \texttt{fillAt} \texttt{fill} rather than draw the oval. Filling an oval includes its outline and its interior. Figure 5.10 shows the same three ovals as given in Figure 5.8, but now filled.

\begin{verbatim}
example = fillAt zero {oval_rx=5, oval_ry=3}
example = fillAt zero {oval_rx=5, oval_ry=5}
example = fillAt zero {oval_rx=3, oval_ry=5}
\end{verbatim}
5.4. **EXAMPLES**

![Three filled oval shapes.](image)

Figure 5.10: Three filled oval shapes.

5.4.5 **Drawing curves**

A Curve is a section of an Oval. A curve is defined by the source oval, curve_oval, a starting angle, curve_from and an ending angle, curve_to, both taken in radians, and the direction in which the section should be taken, curve_clockwise which is a Boolean value. The type definition of a Curve is:

```plaintext
:: Curve
   = { curve_oval :: !Oval
       , curve_from :: !Real
       , curve_to :: !Real
       , curve_clockwise :: !Bool
    }
```

The start and end point of the section are again derived from the unit circle. Given an angle alpha, and a source oval defined by \{oval_rx, oval_ry\}, then the point on the curve (oval) corresponding with alpha is \{x=oval_rx*cos alpha, y=oval_ry*sin alpha\}. If curve_clockwise is True then the section is taken clockwise from the start point to the end point, otherwise it is taken counter clockwise. Figure 5.11 shows two sections of an Oval. In both cases the curve_from angle is \(\frac{\pi}{6}\) and the curve_to angle is \(\frac{3\pi}{2}\). The left section is taken counter clockwise, and the right section is taken clockwise. For your convenience, the value \(\pi\) is approximated in the module StdPictureDef by the macro PI.

```plaintext
curve = { curve_oval = \{oval_rx=5, oval_ry=3\}
         , curve_from = PI/6.0
         , curve_to = 3.0*PI/2.0
         , curve_clockwise = True
      }
example = drawAt zero curve
example = drawAt zero \{curve & curve_clockwise = False\}
```

Figure 5.11 not only shows the curve sections that are taken from an oval, but also what happens when these sections are drawn at a specific position. In both cases the curves are drawn at zero. The starting point, indicated by the starting angle, is determined by the current pen position in case of the draw function of the Drawables type constructor class, and is determined by the Point2 argument of the drawAt function of the Drawables type constructor class.

Drawing a Curve with varying PenSizes is the same as taking the section of the corresponding Oval drawn with that pen size. Figure 5.12 shows three times the same curve taken but drawn with pen sizes 1, 2, and 3 respectively. The source oval
Figure 5.11: Two curves taken clockwise and counter clockwise.

is the same as the one drawn in Figure 5.9. The section is taken counter clockwise from $\frac{3}{4}$ to $\frac{3}{4}\pi$.

\[
oval = \{\text{curve}_\text{oval} = \{\text{oval}_\text{rx}=5, \text{oval}_\text{ry}=5\}, \text{curve}_\text{from} = \pi/4.0, \text{curve}_\text{to} = 1.75*\pi, \text{curve}_\text{clockwise} = \text{False}\}\}
\]

element = \text{drawAt zero oval} o (\text{setPenSize} 1)
element = \text{drawAt zero oval} o (\text{setPenSize} 2)
element = \text{drawAt zero oval} o (\text{setPenSize} 3)

Figure 5.12: Three curves drawn with increasing pen sizes.

Curves are also an instance of the Fillables type constructor class. When filling a curve, the interior formed by the drawn curve and two lines connecting the center of the source oval and the end points of the curve is filled. Figure 5.13 shows the two curves of Figure 5.11, but now using fill rather than draw.

\[
\text{curve} = \{\text{curve}_\text{oval} = \{\text{oval}_\text{rx}=5, \text{oval}_\text{ry}=3\}, \text{curve}_\text{from} = \pi/6.0, \text{curve}_\text{to} = 3.0*\pi/2.0, \text{curve}_\text{clockwise} = \text{True}\}\}
\]

element = \text{fillAt zero curve}
element = \text{fillAt zero} \{\text{curve} \& \text{curve}_\text{clockwise} = \text{False}\}

5.4.6 Drawing rectangles

A Rectangle is a shape of four connected lines that is defined by two diagonally oriented corner Point2s, corner1 and corner2. The type definition of a Rectangle is as follows:
:: Rectangle = {corner1::!Point2, corner2::!Point2}

The Rectangle type constructor is an instance of the Drawables type constructor class. The drawAt function is not very useful because it ignores its Point2 argument and proceeds as draw. Any two Point2s are valid corner points of a Rectangle. In case a Rectangle has a zero width or zero height, drawing that rectangle will show nothing. It does not matter in what order the two corner points are given. This is illustrated by the following Rectangle definitions, displayed in Figure 5.14.

example = draw {zero & corner2={x=10,y=6}}
example = draw {zero & corner2={x=6, y=6}}
example = draw {zero & corner1={x=10,y=6}}

:: Figure 5.13: Two filled curves taken clockwise and counter clockwise.

The shape of a rectangle is also affected by the current PenSize attribute. Increasing the pen size does not increase the outline of the rectangle. The only pixels that are affected are inside the rectangle. Figure 5.15 shows the Rectangle {corner1=zero, corner2={x=10,y=10}} when drawn with pen size 1, 2, and 3.

example = draw {zero & corner2={x=10,y=10}} o (setPenSize 1)
example = draw {zero & corner2={x=10,y=10}} o (setPenSize 2)
example = draw {zero & corner2={x=10,y=10}} o (setPenSize 3)

:: Figure 5.14: Three rectangle shapes.

Rectangles are also an instance of the Fillables type constructor class. The function fill and fillAt fill rather than draw the rectangle. Filling a rectangle includes its outline and its interior. Figure 5.16 shows the same three rectangles as given in Figure 5.14, but now filled.

example = fill {zero & corner2={x=10,y=6}}
example = fill {zero & corner2={x=6, y=6}}
example = fill {zero & corner1={x=10,y=6}}
5.4.7 Drawing boxes

A Box is a Rectangle but without fixing a position. It is therefore only defined by a width, box_w and height, box_h. The type definition of a Box is:

```haskell
:: Box = {box_w :: !Int, box_h :: !Int}
```

The position of a Box is determined by the drawing functions of the type constructor class Drawables. In case of draw, the current pen position is the base point. In case of drawAt, the Point2 argument is the base point. Given this base point base = (x, y), and a box {box_w, box_h}, drawing the box is the same as drawing the rectangle {corner1 = base, corner2 = (x + box_w, y + box_h)}. Any value for box_w or box_h is permitted (so also zero or negative values).

Boxes are drawn and filled in the same way as Rectangles are. So the effect of using different PenSizes is the same as well as filling boxes.

5.4.8 Drawing polygons

A Polygon is an object which shape is formed by a number of Vector2s, such as triangles, rectangles, but also more exotic shapes. The type definition of a Polygon is:

```haskell
:: Polygon = {polygon_shape :: ![Vector2]}
```

A Polygon is always a closed shape. A shape polygon_shape is closed if the following equation holds:

```
foldr (+) zero polygon_shape = zero
```
5.4. EXAMPLES

The object I/O library will always close the `polygon_shape` if this is not the case, so you don’t have to worry about this. Drawing a polygon of shape `polygon_shape` is simply drawing the closed list of vectors in sequence:

```plaintext
seq (map draw polygon_shape)
```

Similar to `Boxes`, `Polygons` do not specify their location. Again, this is determined by the drawing functions of the type constructor class `Drawables`. In case of `draw`, the base point is defined by the current pen position. In case of `drawAt`, the base point is defined by the `Point2` argument. Figure 5.17 shows three polygons defined by the following shapes:

```plaintext
eexample = drawAt zero {polygon_shape=[{vx=8,vy=0},{vx=(-4),vy=8}]}
eexample = drawAt zero {polygon_shape=[{vx=8,vy=0},{vx=0, vy=8}},{vx=(-8),vy=0]}]
eexample = drawAt zero {polygon_shape=[{vx=8,vy=0},{vx=(-8),vy=8}},{vx=8, vy=0]}
```

![Figure 5.17: Three polygon shapes.](image)

Because a polygon is a collection of vectors, its shape is affected by the current `PenSize` attribute. Figure 5.18 shows the three polygons of Figure 5.17 drawn with pen size 2.

![Figure 5.18: Three polygons drawn with pen size 2.](image)

`Polygons` are also an instance of the `Fillables` type constructor class. The function `fill` and `fillAt fill` rather than draw the polygon. Filling a polygon includes its outline and its interior. Figure 5.19 shows the same three polygons as given in Figure 5.17, but now filled.

5.4.9 Drawing bitmaps

Using the drawing operations discussed so far one can produce images that have an ‘algorithmic’ nature: they consist of text, lines, curves, and polygons. Not every
image can be expressed (easily) in this way (consider for instance the image in Figure 5.20). To produce more complex images *bitmaps* are very useful. A bitmap is a prefabricated image of a certain size (in pixels) that is stored in the file system.

![Bitmap Image](image.png)

Figure 5.20: A non algorithmic image.

You can use your favourite drawing package and create and store images as a bitmap. The file format depends on the platform. Currently the following formats are supported:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macintosh</td>
<td>PICT</td>
</tr>
<tr>
<td>Windows(95/NT)</td>
<td>BMP</td>
</tr>
</tbody>
</table>

The bitmap operations can be found in module StdBitmap (Appendix A.1). Bitmaps can be read from file using the function `openBitmap`:

```cpp
openBitmap::! +#Char] !*env -> (!Maybe Bitmap,!*env) | FileSystem env
```

Given the full path name of a bitmap file, `openBitmap` reads the bitmap in memory. If this is successful, *(Just bitmap)* is returned. Reasons for failure are illegal file name arguments, wrong file formats, lack of heap space (in case of Windows(95/NT)), or lack of extra memory (in case of Macintosh)\(^1\).

A Bitmap is an abstract data type. Bitmaps can be drawn in any size you like. For this purpose you can change its size by using `resizeBitmap`. The only information one can retrieve from a Bitmap value is its size.

\(^1\)In the current implementation bitmaps are not garbage collected. This puts a restriction on the number of bitmaps that can be used inside one application. In a future version bitmaps will be garbage collected.
5.4. EXAMPLES

45

resizeBitmap :: !Size !Bitmap -> Bitmap
getBitmapSize :: !Bitmap -> Size

Bitmaps are instances of the Drawables type constructor class. Given a current pen position pos=(x,y) and a bitmap bitmap of size (w,h), the functions (draw bitmap) and (drawAt pos bitmap) both place the bitmap exactly inside the rectangle {corner1=pos, corner2={x=x+w, y=y+h}}.

5.4.10 Drawing in XOR mode

For many programs it is sometimes useful to be able to temporarily draw figures over an existing drawing, and being able to remove them without affecting the source picture. Examples are flashing cursors, selection track boxes, and selection anchor points. For this purpose drawing in XOR mode is supported. The functions appXorPicture and accXorPicture handle this. Both apply their argument function to the argument picture in XOR mode. The latter function also allows its argument function to return a result.

appXorPicture:: !.(IdFun *Picture) !*Picture -> *Picture
accXorPicture:: !.(St *Picture) !*Picture -> (.x,!*Picture)

Drawing in XOR mode has the important property that drawing the same figure twice results in the same picture. Given a drawing function f, the following equations hold:

(appXorPicture f) o (appXorPicture f) = id
(snd o (accXorPicture f)) o (snd o (appXorPicture f)) = id

Let’s explain what happens in the Picture when one uses XOR mode. Consider a source picture, source, shown left in Figure 5.21, which is a circle. Next to the source picture is the picture to be drawn in XOR mode, a fat rectangle, drawn by a drawing function f.

Figure 5.21: A source picture, the figure to be drawn in XOR mode, and the result.

If one interprets the white pixels of both pictures as False, and the black pixels of both pictures as True then the result of drawing f in XOR mode in source is the same as taking the Boolean exclusive or on all such interpreted pixels on the same coordinates. So all pixels that have the same colour become white (False), while all pixels of different colour become black (True). The result of this is shown in the right picture of Figure 5.21. Applying f once more in XOR mode to this new picture yields a picture equal to source.

What happens when using more interesting colours than black and white is basically the same thing. In one way or another, the exclusive or is taken from the source.
picture and the drawing operations in such a way that repeating it gives the source picture again. What the colours of the ‘xor-ed’ picture are depends on the platform, and is not specified by the object I/O library.

5.4.11 Drawing in Hilite mode

Programs that want to indicate selections (for instance text segments in a word processor, or image components in a drawing program) can do this by drawing the selected area in *hilite* mode. For this purpose the type constructor class *Hilites* is used. Its type definition is:

```
class Hilites figure where
    hilite :: !figure !*Picture -> *Picture
    hiliteAt :: !Point2 !figure !*Picture -> *Picture
```

The instances of *Hilites* are *Box* and *Rectangle*. The pixels that are affected by *hilite* and *hiliteAt* are the same as for *fill* and *fillAt*. Drawing in hilite mode has the same property as drawing in *xOR* mode that drawing the same figure twice on a source picture leaves the source picture unchanged. Given a figure *figure*, the following equations hold:

\[(\text{hilite } \text{figure}) \circ (\text{hilite } \text{figure}) = \text{id}\]
\[(\text{hiliteAt } \text{figure}) \circ (\text{hiliteAt } \text{figure}) = \text{id}\]

The visual effect of hiliting these areas depends on the platform. On some platforms hiliting an area will change the colour of all pixels that have the background colour to a special hilite colour, ignoring all other pixels. Hiliting the area once more will revert the hilite colours back to the background colour. If a platform does not support hilite mode, the area will be drawn in *xOR* mode (shown in Figure 5.22). The source picture is the text "Pop" of Figure 5.7.

```
example = hilite {corner1={x=0,y=2},corner2={x=24,y=10}}
    o (drawAt zero "Pop")
```

![Figure 5.22: Hiliting a rectangular area using *xOR* mode.](image)

5.4.12 Drawing in Clipping mode

Drawing in *clipping* mode is a powerful technique to create graphics that can not be drawn (or using much more complicated expressions) using only the drawing mode.
primitives discussed before. In clipping mode, the programmer specifies a *region* that works like a mask: drawing proceeds as described above, but only those pixels that are inside the clipping region are actually drawn.

A region is an abstract data type, `Region`. Regions are created by composing `Rectangles` and `Polygons`, using the type constructor class `toRegion`. When composing regions, using the list and `&` instance, the union of the argument regions is taken.

class toRegion area :: !area -> Region

:: PolygonAt
   = { polygon_pos :: !Point2
        , polygon :: !Polygon
     }

instance toRegion Rectangle
instance toRegion PolygonAt
instance toRegion [r]      | toRegion r
instance toRegion [: r1 r2] | toRegion r1 & toRegion r2

Clipping is done using the functions `appClipPicture` and `accClipPicture`:

appClipPicture:: !Region !(IdFun *Picture)!Picture->[Picture
accClipPicture:: !Region !(St *Picture .x)!Picture->(x,!*Picture)

Suppose we have the following drawing function, `f`, which draws a number of horizontal lines with a result as shown in Figure 5.23:

```haskell
f = seq [drawLine {x=0,y=y} {x=9,y=y} \ y<-[0,2..8]]
```

![Figure 5.23: The source picture.](image)

As clipping regions the polygons shown in Figure 5.17 are used. Figure 5.24 shows the result of the following clipping functions:

```haskell
example = appClipPicture
  (toRegion { polygon_pos =zero
           , polygon_shape= [[vx=8,vy=0},${vx=(-4),vy=8}]
      } ) f
example = appClipPicture
  (toRegion { polygon_pos =zero
             , polygon_shape= [[vx=8,vy=0},${vx=0, vy=8}]
             } ) f
```
Figure 5.24: The clipped source picture.
Chapter 6

Windows and dialogues

In most graphical user interfaces the major top level interactive objects a user is confronted with are windows and dialogues. This is also the case for the object I/O library. Windows and dialogues share the same set of controls and many library functions. Dialogues are considered to be specialised windows. Depending on the platform, dialogues usually offer a special, enhanced, user interface. Another difference is that dialogues can be opened modally. In this mode the user is forced by the program to handle the dialogue completely before continuing with the program. To emphasize the similarity between windows and dialogues, their algebraic type definitions are almost identical (these can be found in module StdWindowDef):

\[
\text{:: Window c ls pst} \\
\quad = \text{Window Title (c ls pst) [WindowAttribute *(ls,pst)]} \\
\text{:: Dialog c ls pst} \\
\quad = \text{Dialog Title (c ls pst) [WindowAttribute *(ls,pst)]}
\]

Before delving into details, we first introduce basic terminology for windows and dialogues in Section 6.1 and discuss their attributes in Section 6.2. Opening and closing of windows and dialogues is handled in Section 6.3. We then discuss the ways to handle the components windows and dialogues are made of in sections 6.4, 6.5, and 6.6. Handling keyboard and mouse input is presented in Section 6.7. Finally, this chapter is concluded with a treatment on modal dialogues in Section 6.8.

6.1 Basic terminology

The main purpose of a window is to present to the user a view on a document, graphically represented as an object of type Picture. Pictures have been discussed in Chapter 5. By using the mouse and keyboard, the user can manipulate the document. Controls in a window can add further manipulation functionality. The main purpose of a dialogue is to present to the user a structured way of passing information to perform actions. This structured communication is realised by means of controls.

6.1.1 Anatomy of windows and dialogues

Although from a user’s perspective windows and dialogues appear to be ‘solid’ objects (Figure 6.1) it is illustrative to have a look at a window from a different perspective.
CHAPTER 6. WINDOWS AND DIALOGUES

A window is composed of three layers, see Figure 6.2. The bottom layer, the document layer, is formed by the rendered document, the Picture. The middle layer, the control layer, contains all controls of the window. The top layer, the window frame, typically consists of a title bar, and window components to close and resize the window. The window frame can have any size and is restricted only by the screen size and a platform dependent minimum size. The window frame serves as a clipping area of the document layer. Windows usually contain scrollbars to help the user change the current view on the document layer. The current part of the document layer that is visible is called the view frame. The default drawing domain of the document layer Picture ranges from 0 to $2^{30}$ in both axes. This is in general too large for rendering the document. A window can limit the displayable range of a window by setting its view domain.

In contrast with windows, a dialogue is composed of only two layers, the control layer and the window frame, or also called the dialogue frame. Instead of a document layer, a dialogue has a platform dependent background. The program can not draw
into nor navigate the background. The dialogue frame can not be resized by the user and is usually big enough to display the whole control layer.

For both windows and dialogues, the programmer has full control over the view frame, control layer, and document layer. Section 6.4 discusses how to handle the document layer. Handling the control layer is discussed in Section 6.5. The window and dialogue frame are controlled by the platform, see Section 6.6.

6.1.2 Stacking order

In general, a program can have an arbitrary number of windows and dialogues opened at the same time. These elements appear in a *stacking order*, seen by the application user in top to bottom order. For normal windows and dialogues the stacking order is not fixed. *Modal* dialogues however always appear topmost. Windows and modeless dialogues that are opened while (several) modal dialogues are open always appear below the bottom most modal dialogue.

6.1.3 Active window or dialogue

When a user works with a program, exactly one window or dialogue receives all keyboard and mouse input. This element is called the *active* window/dialogue and it has the *input focus*. With the exception of modal dialogues, the active window/dialogue does not necessarily occupy the top most stacking position.

6.2 Window and dialogue attributes

WindowAttribute is the type of window and dialogue attributes. The table below shows which attributes are valid for which element.

<table>
<thead>
<tr>
<th>Window attributes</th>
<th>For windows and dialogues:</th>
<th>For windows only:</th>
<th>For dialogs only:</th>
</tr>
</thead>
<tbody>
<tr>
<td>WindowActivate</td>
<td></td>
<td>WindowCursor</td>
<td>WindowCancel</td>
</tr>
<tr>
<td>WindowClose</td>
<td></td>
<td>WindowHScroll</td>
<td>WindowOk</td>
</tr>
<tr>
<td>WindowDeactivate</td>
<td></td>
<td>WindowKeyboard</td>
<td></td>
</tr>
<tr>
<td>WindowHMargin</td>
<td></td>
<td>WindowLook</td>
<td></td>
</tr>
<tr>
<td>WindowId</td>
<td></td>
<td>WindowMouse</td>
<td></td>
</tr>
<tr>
<td>WindowIndex</td>
<td></td>
<td>WindowOrigin</td>
<td></td>
</tr>
<tr>
<td>WindowInit</td>
<td></td>
<td>WindowPen</td>
<td></td>
</tr>
<tr>
<td>WindowInitActive</td>
<td></td>
<td>WindowSelectState</td>
<td></td>
</tr>
<tr>
<td>WindowItemSpace</td>
<td></td>
<td>WindowViewDomain</td>
<td></td>
</tr>
<tr>
<td>WindowOuterSize</td>
<td></td>
<td>WindowVScroll</td>
<td></td>
</tr>
<tr>
<td>WindowPos</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WindowViewSize</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WindowVMargin</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.2.1 Window and Dialog attributes

WindowActivate and WindowDeactivate

These attributes define the behaviour of the window in case the window becomes the active window (WindowActivate), and is no longer the active window (WindowDeactivate) respectively (see also 6.1.3). If no attribute is provided, this information will not be passed to the program.
WindowClose
This attribute adds the platform dependent close facility to the window/dialogue. The associated function will be evaluated in case this control is triggered. Actually closing the window/dialogue is the responsibility of this function. In case no WindowClose attribute is provided, the window/dialogue cannot be closed in that way.

WindowHMargin and WindowVMargin
These attributes determine the left-right, and top-bottom margin of a window/dialogue respectively. Their default value in case of windows is zero, and platform dependent for dialogues.

WindowId
This attribute identifies the window/dialogue to which it is associated. If you do not provide a WindowId, the object I/O system open function creates a fresh Id for the window/dialogue.

WindowIndex
This attribute determines the initial stacking position of the window/dialogue (see also 6.1.2). If no WindowIndex attribute is provided, then the window/dialogue will be opened frontmost. Modal dialogues are always opened frontmost.

WindowInit
This attribute defines an action that should be performed immediately after opening the window/dialogue. This is equivalent to the process initialisation action (see Section 2.3 and Chapter 11). If no WindowInit attribute is provided, no additional action is performed.

WindowInitActive
This attribute defines which control should have the initial input focus when its parent window/dialogue is opened. If no WindowInitActive attribute is provided, the first keyboard sensitive control is selected.

WindowItemSpace
This attribute determines the space between controls if no further offsets are provided in the layouts of controls. The default values are identical for windows and dialogues.

WindowOuterSize and WindowViewSize
These two attributes determine the initial size of the window/dialogue. The first attribute defines the size in terms of the window/dialogue frame size. The second attribute defines the size in terms of the window/dialogue view frame size. If neither attribute is provided, then the system will derive a proper size. In case of dialogues the size is determined by the set of controls, margins, and item spaces. In case of windows the screen size is chosen.

WindowPos
This attribute determines the initial position of the window/dialogue (see also Section 7.3). Relative Ids that occur in the ItemPos refer to other windows/dialogues. The object I/O system will always place a window/dialogue visibly on the current screen. If you do not provide a WindowPos to a window, then the window will be placed at the left top of the screen. If you do not provide a WindowPos to a dialogue, then the dialogue will be placed at a position conform the platform user interface (typically centered).
6.2.2 Window attributes

WindowCursor
This attribute defines the shape of the cursor in case the mouse is over the window and not inside a control that may overrule this shape. In case no attribute is provided moving the mouse over the window will not change its shape.

WindowHScroll and WindowVScroll
These attributes add a horizontal scrollbar and a vertical scrollbar to the window. If no attribute is given, no scrollbars are added.

WindowKeyboard and WindowMouse
These attributes allow a window to respond to user actions with the mouse (WindowMouse) and keyboard (WindowKeyboard). If no attribute is provided, then this information will not be passed to the program. Both attributes can define an additional filter to ignore some input actions. If the SelectState of the window is Unable then neither function will obtain input.

WindowLook
This attribute defines a function that, given the current SelectState of the window and information about which part of the window should be displayed, defines what the window should look like. The default look fills the window with the platform dependent background colour. (The Look function plays a similar role for CustomButtonControls (Section 7.1.3), CustomControls (Section 7.1.4), and CompoundControls (Section 7.1.11. It can also be used for printing (Chapter 13)).

In addition to the Look function, the WindowLook attribute also has a boolean argument. If this attribute is True, then the object I/O system assumes that the content of the window is independent of the actual view frame orientation. In that case scrolling and changing the size of the window can be done more efficiently. If the attribute is False, then the Look function is always applied to the whole current view frame. See also Section 6.4.1 for more information on using this attribute.

WindowOrigin
This attribute determines the initial position of the ViewFrame, the rectangular part of the ViewDomain that is currently visible in the window. This position is always verified to be within the given ViewDomain. If no WindowOrigin attribute is provided, then the left-top coordinates of the ViewDomain are used.

WindowPen
This attribute determines the initial pen that is used to draw in the Picture of the window. If no WindowPen attribute is provided, then the default pen attributes are used (see also Section 5.1).

WindowSelectState
This attribute defines whether the window accepts user keyboard or mouse input (Able) or not (Unable). The default value is Able. Note that although a window can be active, it can also be disabled. This only means that all input is ignored by the window.

WindowViewDomain
This attribute defines the drawing coordinate system of the document layer (see also 6.1.1). Drawing operations outside this area will be clipped. If
no ViewDomain is provided, then the window will obtain the ViewDomain \{viewDomainRange & corner1=zero\}.

### 6.2.3 Dialogue attributes

**WindowCancel and WindowOk**

These attributes indicate which control should act conform the platform user interface ‘cancel’ control and ‘confirm’ control respectively. If such an attribute is not provided, then no control is selected.

### 6.3 Opening and closing of windows and dialogues

Windows and dialogues are opened using the appropriate instances of the respective type constructor classes `Windows` and `Dialogs` (module `StdWindow`, Appendix A.47). Windows and dialogues share the same set of controls. To enforce this by the type system, the instance declarations of windows and dialogues are provided with a type constructor class restriction on the type constructor variable `c`, which must be an instance of the `Controls` type constructor class.

```haskell
class Windows wdef where
    openWindow :: !ls !(wdef .ls (PSt .l)) !(PSt .l)
        -> (!ErrorReport, !PSt .l)

...;
```

```haskell
class Dialogs ddef where
    openDialog :: !ls !(ddef .ls (PSt .l)) !(PSt .l)
        -> (!ErrorReport, !PSt .l)

...;
```

```haskell
instance Windows (Window c) \| Controls c
instance Dialogs (Dialog c) \| Controls c
```

To close windows and dialogues the function `closeWindow` has to be used. In many contexts one wants to close the active window or dialogue. For these cases the convenience function `closeActiveWindow` is provided.

```haskell
closeWindow :: !Id !(PSt .l) -> PSt .l
closeActiveWindow :: !!(PSt .l) -> PSt .l
```

A typical example of using `closeActiveWindow` is the callback function for the `WindowClose` attribute. If it is allowed to close the window or dialogue under all circumstances (nothing needs to be saved for instance) then the following attribute definition is sufficient:

```haskell
WindowClose (noLS closeActiveWindow)
```

Note that `noLS` is used to obtain a function of the requested type \((ls, PSt .l) \to (ls, PSt .l)\).
6.4 Handling the document layer

The document layer of a window is used to present visual feedback to the user on the current status of the document that is being manipulated. Only windows have a document layer. Below we discuss two ways of drawing into the document layer. The first method, indirect rendering (Section 6.4.1), uses the look function, the second method, direct rendering (Section 6.4.2), draws directly into the document layer. Some programming pragmatics are discussed in Section 6.4.3. We conclude with an example in Section 6.4.4.

6.4.1 Indirect rendering

Two WindowAttributes play a paramount role with respect to the document layer: WindowViewDomain and WindowLook. Let’s have a closer look at them. Here are their type definitions (in module StdWindowDef):

```plaintext
:: WindowAttribute st
 = ... | WindowLook Bool Look
       | WindowViewDomain ViewDomain
       | ...
```

The document layer is rendered using a Picture. As we have seen in Section 6.2, the default drawing range of a Picture is \((0, 2^{30})\) in both axes. This range can be changed by the WindowViewDomain attribute. It has a ViewDomain argument which is defined as a Rectangle:

```plaintext
:: Rectangle
 = \{ corner1 :: !Point2, corner2 :: !Point2 \}
:: Point2
 = \{ x :: !Int , y :: !Int \}
```

A Rectangle is a record consisting of the two diagonally opposite corner points of the new drawing range. The view domain can have any value that is between the `viewDomainRange` macro defined in StdIOCommon:

```plaintext
viewDomainRange ::= \{ corner1 = \{ x = 0-(2\cdot30), y = 0-(2\cdot30)\} , corner2 = \{ x = 2\cdot30, y = 2\cdot30 \} \}
```

The only illegal value is to have identical x or y coordinates for the corner points of a view domain. It causes your application to abort with the message “Error in rule validateWindowDomain [windowvalidate]: Window has illegal ViewDomain argument”.

The view domain defines the coordinates for your window to draw into. If a window has a view domain that is smaller than the available screen estate, it is possible that the size of a window exceeds the size of its view domain. So the view frame of a window may be larger than the view domain of the window. It is allowed to draw in the extra area as well. The coordinate system simply extends to the right and downwards. A window never displays an area that lies to the left or above the minimum view domain values.

The Picture of the document layer is rendered using the WindowLook attribute. This attribute has a Look function argument. It is defined as follows:
Whenever it is necessary to render (part of) the visible document layer, the object I/O system will apply the look function of the WindowLook attribute. It will be parameterised with the current SelectState of the window and detailed information about which part of the current view frame needs to be rendered. This information is presented by means of the UpdateState record:

```haskell
:: UpdateState
  = { oldFrame :: !ViewFrame
      , newFrame :: !ViewFrame
      , updArea :: !UpdateArea
      }
:: ViewFrame ::= Rectangle
:: UpdateArea ::= [ViewFrame]
```

The three fields of the UpdateState record contain the following information:

**oldFrame**
If the size or orientation of the window was changed, then this field contains the view frame before that change. If this is not the case, then this field contains the current view frame.

**newFrame**
If the size or orientation of the window was changed, then this field contains the view frame after that change (so, the current view frame). If this is not the case, then this field contains the current view frame.

**updArea**
This field contains a list of Rectangles (not necessarily disjoint) that define the parts of the visible current view frame that need to be drawn. This list always consists of at least one element. Each of its elements is always completely inside the current view frame. The remark that the rectangles need not necessarily be disjoint is important in case one draws in XOR or palette mode (see Sections 5.4.10 and 5.4.11). Drawing in overlapping areas using one of these modes results in undoing the intended effect.

Given the current SelectState of the window and the UpdateState, the look function is applied to the current Picture of the document layer. For this reason, the look function must be written in such a way that it works correctly for any argument Picture. You can not make any assumptions on the current content of the update area rectangles except that they are very unlikely to be correct.

In addition to the Look function, the WindowLook attribute has a Bool argument. If this value is False, then you have full control over drawing in a window. For instance, when the size of a window is decreased, the UpdateState passed to the Look will contain in its oldFrame the larger view frame before resizing, and in newFrame the decreased view frame after resizing. In addition, the updArea will consist of one rectangle only, equal to newFrame. In this way you can let the content of the image depend on the current window view frame. The following Look function draws a box the size of its new view frame and its two diagonal lines (see figure 6.3):

```haskell
crossbox _ {oldFrame,newFrame} picture
```
6.4. HANDLING THE DOCUMENT LAYER

```haskell
# picture = seq (map undraw (crossboxlines oldFrame)) picture
# picture = seq (map draw (crossboxlines newFrame)) picture
= picture

crossboxlines {corner1=a,corner2=b}
= [ {line_end1=p1,line_end2=p2}
    \ (p1,p2)\in\{(a,c),(a,d),(a,b),(b,c),(b,d),(c,d)\}
  ]
where c = {b & y=a.y}; d = {a & y=b.y}

Figure 6.3: Letting the Look function follow the window size.

The purpose of the WindowLook attribute function is to describe the look of the current state of the document layer. If the document does not change, then this function always correctly renders the document. However, for most windows the state of the document changes during its life-cycle. The WindowLook attribute can be changed and retrieved using the StdWindow functions setWindowLook and getWindowLook:

```haskell
setWindowLook :: !Id !Bool !(IOSt .l) -> IOSt .l
getWindowLook :: !Id !(IOSt .l)
    -> (!Maybe (Bool,Look),!IOSt .l)
```

`setWindowLook` changes the current WindowLook attribute of the window indicated by the `Id` argument with the new `(Bool,Look)` argument, provided this window is present and does not refer to a dialogue. If the first `Bool` argument is `False` then that's all. If it is `True`, then the look function will be applied to the window in the way described above.

6.4.2 Direct rendering

Instead of using only the Look function of a window, one can also draw directly in the Picture of the window's document layer. This is done using the functions appWindowPicture and accWindowPicture (StdWindow, Appendix A.47):

```haskell
appWindowPicture:: !Id !(IdFun *Picture) !(IOSt .l) -> IOSt .l
accWindowPicture:: !Id !(St *Picture .x) !(IOSt .l)
    -> (!Maybe .x,IOSt .l)
```
appWindowPicture applies the argument drawing function to the Picture in the document layer of the window indicated by the Id argument if this window is present and does not refer to a dialogue. accWindowPicture is the same, except that its argument function returns a result of some type. If the indicated window was indeed found, and the argument function returned a value x, then the result of accWindowPicture is (Just x). In all other cases it is Nothing.

For some visual feedback such as drawing blinking cursors or track boxes this method is more suitable than the indirect way of using the Look function. The timer example in Section 9.1.1 shows a window in which the direct rendering method is applied.

### 6.4.3 Pragmatics

The WindowLook function is used by the object I/O system for all cases that the content of the window needs to rendered. Among others, causes are when the view frame, size, stacking order, or selectstate of a window changes. It is very annoying for the application user when these actions take to much time. Therefore it is worth your while to spend some effort in getting a good performance out of the list of drawing functions.

### 6.4.4 Example: displaying a bitmap

In this example we create a program that allows the user to select an arbitrary bitmap which will be displayed in a window (see Figure 6.4).

The first thing to do is to give the user the opportunity to select a bitmap. For this purpose the library function selectInputFile in module StdFileSelect (Appendix A.10) is provided. Applying this function opens the platform standard file selector dialogue. When a file has been selected by the user the return value contains the full pathname of the selected file. If no file has been selected, then Nothing is returned. This function belongs to the FileSelectEnv type constructor class which contains two other functions: selectOutputFile (which allows a user to select a (new) output file) and selectDirectory (which allows a user to select a directory).

```haskell
class FileSelectEnv env where
  selectInputFile :: env -> (Maybe String, env)
  selectOutputFile :: String String env -> (Maybe String, env)
  selectDirectory :: env -> (Maybe String, env)
```
The first part of the program looks as follows (if no file was selected the program simply terminates):

Start :: *World -> *World
Start world
  # (maybeFile,world) = selectInputFile world
  | isNothing maybeFile
  = world

Given a selected input file pathname, the function openBitmap (StdBitmap, Appendix A.1) can be used to read in the bitmap (see also Section 5.4.9).

openBitmap:: ![#Char] ![*env-> (!Maybe Bitmap,!*env) | FileSystem env

openBitmap returns a bitmap if it could be read in successfully, otherwise it returns Nothing (in that case we also let the program terminate). This part of the program is as follows:

  # bitmapfile = fromJust maybeFile
  # (maybeBitmap,world) = openBitmap bitmapfile world
  | isNothing maybeBitmap
  = world

Having successfully read in the bitmap, we can determine its size using the StdBitmap function getBitmapSize.

  # bitmap = fromJust maybeBitmap
  bitmapsize = getBitmapSize bitmap

The window that displays the bitmap has the same initial size as the bitmap (setting the (WindowViewSize bitmapsize) attribute). In this example we use the indirect rendering method. This is done by setting the (WindowLook True \(_ _ -> drawAt zero bitmap) attribute. Finally, when the user requests closing of the window, the application simply terminates. This is done by setting the (WindowClose (noLS closeProcess)) attribute. So we obtain the following window definition:

```
window = Window "Bitmap" NilLS
    [ WindowViewSize bitmapsize
    , WindowLook True \(_ _ -> drawAt zero bitmap)
    , WindowClose (noLS closeProcess)
    ]
```

The final step is to create an interactive process that contains the window by applying the startIO function (StdProcess, Appendix A.27). The only initialisation action of the process is to open the window. In case the user requests the process to terminate, the application obeys. This is done by setting the (ProcessClose closeProcess) attribute.

```
= startIO SDI
  Void
  (snd o openWindow Void window)
  [ProcessClose closeProcess]
  world
This completes the program. Here is the complete program code.

```haskell
module showbitmap

// **********************************************************************************
// Clean tutorial example program.
// // This program creates a window that displays a user selected bitmap.
// // Make sure that this application has sufficient heap or extra memory.
// **********************************************************************************

import StdIO, StdEnv

Start :: *World -> *World
Start world

# (maybeFile,world) = selectInputFile world
| Nothing maybeFile = world
# bitmapfile = fromJust maybeFile
# (maybeBitmap,world) = openBitmap bitmapfile world
| Nothing maybeBitmap = world
| otherwise
# bitmap = fromJust maybeBitmap
bitmapsize = getBitmapSize bitmap
window = Window "Bitmap" NilLS
  [ WindowViewSize bitmapsize
  , WindowLook True (\_ -> drawAt zero bitmap)
  , WindowClose (noLS closeProcess)
  ]
  = startIO SDI

Void
  (snd o openWindow Void window)
  [ProcessClose closeProcess]

world

6.5 Handling the control layer

The control layer contains the controls of a window or dialogue. Windows and
dialogues can have the same set of controls. As we have seen in Section 6.3, this
has been made explicit by the type constructor class instance declarations of the
respective type constructor classes.

The standard set of Controls instances is defined in the module StdControlClass
(Appendix A.6). Many operations with respect to the control layer are identical to
operations on controls that are element of CompoundControls. Controls and their
operations are discussed in detail in Chapter 7.

6.6 Handling the window and dialogue frame

The window and dialogue frame are the ‘physical’ borders of windows and dialogues.
A user can grab them using the mouse or some keyboard interface and drag them
around, change the size (in case of windows), or dispose of them. The program
has very limited influence on both the appearance and functionality of the frame.
When opening a window or dialogue, the WindowAttributes are important. This
is discussed in Section 6.6.1. User actions on the window or dialogue frame are
handled by the program via the callback function mechanism. The program can
also change frame properties. This is discussed in Section 6.6.2.
6.6.1 Opening a window or dialogue frame

The attributes of a window and dialogue definition that influence the window and dialogue frame are of course the Title and the following WindowAttributes:

WindowOuterSize and WindowViewSize

These attributes give the preferred size of the window or dialogue frame and the preferred size of the window or dialogue view frame respectively. By convention, if the attribute list of a window or dialogue contains both attributes, then only the first of these is valid.

WindowClose

If this attribute is present, then the window or dialogue frame is provided with a platform dependent interface element to allow the user to request the program to close that window or dialogue.

WindowHScroll and WindowVScroll

Although the horizontal and vertical scrollbar of a window are element of the control layer (Figure 6.2), their behaviour is intimately connected with the size of the view frame and therefore also with the size of the window and dialogue frame.

6.6.2 Changing a window and dialogue frame

The two most apparent changes to the window or dialogue frame are changes of orientation and size. In case of windows, these can be caused by the application user, depending on the attributes as explained in Section 6.6.1.

The program can also change the size of the frame for both windows and dialogues. For dialogues this can be done only indirectly by opening or closing controls. For windows this can also be done directly. If the program changes the view frame (either its size or orientation) then the layout of controls is changed in the same way as if the user had caused this change.

6.7 Handling keyboard and mouse input

Of all windows and dialogues that are in control by an application, at most one receives the keyboard and mouse input. This window is the active window.

Except when modal dialogues are open, the application user can always select one of the visible windows or dialogues to become the new active window. Windows and dialogues can be notified of these events by adding two WindowAttributes: WindowActivate and WindowDeactivate. Both attributes are parameterised with a function that will be applied by the object I/O system as soon as that window becomes active or has become inactive respectively. It is guaranteed that the WindowDeactivate attribute function is applied before the WindowActivate function.

The underlying platform always gives visual clues to the application user about which window or dialogue is currently active. The program can retrieve this information using the getActiveWindow function (Appendix A.47). One should be aware that it is not correct to assume that the active window or dialogue has the top-most stack order position. As an example, one might try to get the Id of the active window by taking the Id from the first element of the result list of getWindowStack, but this only returns the top most window and not the active window.
The program can also activate windows and dialogues. This is done with the `setActiveWindow` function (Appendix A.47). Because modal dialogues are always front-most and the front-most modal dialogue is active, one cannot activate a window or modeless dialogue while modal dialogues are open. Instead, `setActiveWindow` restacks such a window or dialogue immediately behind the bottom-most modal dialogue without making it the active window.

The active window receives all keyboard and mouse input. If this window contains controls it can be the case that the input is channelled to one of these controls. That particular control then has the input focus. If no control has the input focus, then all input is handled by the active window. In case the active window is a dialogue its response to input is defined entirely by the underlying platform. In case of windows the program can customise the behaviour by adding a `WindowKeyboard` attribute for keyboard input (Section 6.7.1) and by adding a `WindowMouse` attribute for mouse input (Section 6.7.2). Both attributes have a filter function (`KeyboardStateFilter` and `MouseStateFilter` respectively) which is applied before the actual callback function is evaluated. Only if the filter returns True then the callback function is evaluated.

### 6.7.1 Keyboard input

Every keyboard sensitive interface object has a `KeyboardFunction` which is a process state transition function that receives, as a first argument, a value of type `KeyboardState`. This value represents one keyboard event. Keyboard events are always generated in sequences that are characterised by a value of type `KeyState` in the following order:

\[
\text{(KeyDown False)} \rightarrow \text{(KeyDown True)} \rightarrow \text{KeyUp}
\]

A keyboard event concerns either the input of an ASCII character (CharKey alternative) or a special key (SpecialKey alternative). The alternative KeyLost is generated whenever the sequence above was interrupted for some reason (for instance, when another window has become active).

```haskell
:: KeyboardState
= CharKey Char KeyState
| SpecialKey SpecialKey KeyState Modifiers
| KeyLost
```

The SpecialKeys are imported via the module `StdKey` (Appendix A.15). Among others they define the function keys, arrow keys, page and line keys. The Modifiers type is defined in `StdIOCommon`. It is a record that refers to the state of the meta keys of the keyboard. Because some ASCII characters are generated using these meta keys they are not provided at the CharKey alternative. So pressing `shift` `a` simply generates the value (CharKey `'A'` (KeyDown False)).

The object I/O system guarantees that at all times only one keyboard alternative is being handled. Assume that a user is pressing the `a` key on the keyboard. This generates a character `a` key down event (CharKey `'a'` (KeyDown False)), and then a sequence of character `a` repeat key events (CharKey `'a'` (KeyDown True)).

If the user now also presses the `b` key, the object I/O system inserts two virtual events that force the program to believe that the user first released the `a` key with a character `a` key up event (CharKey `'a'` KeyUp), and then pressed the `b` key with a character `b` key down event (CharKey `'b'` (KeyDown False)). These are followed by character `b` repeat key events.
Example: keyspotting

To illustrate the use of keyboard handling we create a program that has a window in which the last keyboard input is displayed. Figure 6.5 presents a snapshot of the program.

To allow the window to track keyboard input, it must have the WindowKeyboard attribute. Because we intend to monitor every keyboard input the attribute’s keyboard filter function argument must accept every KeyboardState. This can be defined conveniently by \((\text{const True})\), using the StdFunc library function \text{const}. Of course the keyboard function must handle keyboard input, so the SelectState attribute is \text{Able}. The keyboard function, keyspotting is not interested in the local state of the window which can be ignored by using the lifting function \(\text{noLS1}\). The keyboard function is parameterised with the Id of the window, \(\text{wid}\). This gives us the following definition of the WindowKeyboard attribute:

\[
\text{WindowKeyboard (const True) Able (noLS1 (spotting wid))}
\]

The keyboard function \text{spotting} uses the indirect rendering method (discussed in Section 6.4.1) to display the last keyboard input. It applies the \text{setWindowLook} function (\text{StdWindow}, Appendix A.47) to change the Look function of the window each time new keyboard input reaches the window. For this reason, \text{spotting} is parameterised with the Id of the window. In \text{StdIOLCommon} for all type definitions that do not contain function types an instance is defined for \text{toString}. The type of \text{spotting} can be generalised so that it works for any second argument for which an instance of the overloaded function \text{toString} exists.

\[
\text{spotting :: Id x (PSt .1) -> PSt .1 | toString x}
\]

\[
\text{spotting wid x pst}
\]

\[
= \text{appPIO (setWindowLook wid True (False,look (toString x))) pst}
\]

The Look function of the window is parameterised with the string that should be displayed. It simply centers this string in the current view frame. The size of the view frame (a value of type \text{Rectangle}) can be determined using the \text{StdIOBasic} function \text{rectangleSize}. The size of the string, when drawn with the current pen, can be determined using the \text{StdPicture} function \text{getPenFontStringWidth} (see also Section 5.3).

\[
\text{look :: String SelectState UpdateState *Picture -> *Picture}
\]

\[
\text{look text _ (newFrame) picture}
\]

\[
\begin{align*}
\# \text{picture} &= \text{unfill newFrame picture} \\
\# \text{(width,picture)} &= \text{getPenFontStringWidth text picture}
\end{align*}
\]
= drawAt \{x=(w-width)/2,y=h/2\} text picture
where
\{w,h\} = rectangleSize newFrame

Setting up look in this way has the advantage that the window’s appearance adapts itself to its actual size.

The only things that need to be done to get a complete program is to create an Id value for the window, and start an interactive process that opens the window. This is shown below in the complete code of the keyspotting example.

```hs
module keyspotting

// ****************************************
// Clean tutorial example program.
// This program monitors keyboard input that is sent to a Window.
// ****************************************
import StdEnv,StdIO

Start :: *World -> *World
Start world
  # (wid,world) = openId world
  # window = Window "keyspotting" nils
      [ WindowKeyboard (const True) Able (noLS1 (spotting wid))
        , WindowId wid
        , WindowClose (noLS closeProcess)
      ]
  # startIO SDI
  Void
  [snd o openWindow Void window]
  [ProcessClose closeProcess]
world

where
spotting :: Id x (PSt .l) -> PSt .l | toString x
spotting wid x pst
  = appPIO (setWindowLook wid True (False,look (toString x))) pst

look :: String SelectState UpdateState *Picture -> *Picture
look text _ (newFrame) picture
  # picture = unfill newFrame picture
  # (width,picture) = getPenFontStringWidth text picture
  = drawAt \{x=(w-width)/2,y=h/2\} text picture

6.7.2 Mouse input

Every mouse sensitive interface object has a MouseFunction which is a process state transition function that receives, as a first argument, a value of type MouseState. This value represents one mouse event.

:: MouseState
  = MouseMove Point2 Modifiers
  | MouseDown Point2 Modifiers Int
  | MouseDrag Point2 Modifiers
  | MouseUp Point2 Modifiers
  | MouseLost

The Point2 and Modifiers types are defined in the modules StdIOBasic and StdIOCommon respectively. The Point2 argument gives the position of the mouse
in terms of the view domain coordinates of the interactive object that contains the specific MouseFunction. The Modifiers type constructor is a record that refers to the state of the meta keys of the keyboard that were pressed at the mouse event.

Mouse events are always generated in sequences that are characterised by the alternative constructor of the MouseState type constructor:

```
{MouseMove}* MouseDown {MouseDrag}* MouseUp
```

The Int argument of the MouseDown alternative gives the number of times the mouse was down within the mouse double down time. The mouse double down time is a platform dependent time interval that distinguishes two sequential mouse down events from a double click event. Although an integer is used for this count, its maximum value is usually three. If a mouse down event with count \(i\) has occurred and a new mouse down event is generated within the mouse double down time, then the next mouse down event has count \(i + 1\). If the next mouse down event is not generated within the mouse double down time, then the next mouse down event has count 1.

The MouseLost alternative is generated whenever for some reason the sequence above is interrupted (for instance when another window has become active).

The object I/O system guarantees that every MouseFunction of a mouse sensitive interface object is applied to a sequence of mouse events as characterised above. Assume that a certain window is active and the user is pressing the mouse. This generates first a mouse down event (MouseDown alternative), followed by a sequence of mouse drag events (MouseDrag alternative). If for some reason another window is being activated, the object I/O system inserts a virtual event that forces the program to believe that the user has released the mouse button with a mouse up event (MouseUp alternative). If the new window is also mouse sensitive, then its MouseFunction is applied to a new virtual event that forces the program to believe that the user has pressed the mouse again with a mouse down event (MouseDown alternative). These are followed again by mouse drag events.

**Example: mousespotting**

To illustrate the use of mouse handling we create a program that monitors the mouse input of a window (see Figure 6.6). This program is almost identical to the keyspotting example in the Section 6.7.1. The only differences are the title of the window and the replacement of the WindowKeyboard attribute by a WindowMouse attribute.

Here we take advantage of the fact that the spotting function is overloaded. For completeness, the mousespotting example is shown here.

```haskell
module mousespotting

// Clean tutorial example program.

import StdEnv, StdIO

Start :: *World -> *World
Start world
    # (wid,world) = openId world
    # window = Window "mousespotting" NilLS
        [ WindowMouse (const True) Able (notLS (spotting wid))
```

```haskell```
Windows and dialogues have many aspects in common. One important difference, which was also mentioned in the introduction of this chapter, is that only dialogues can be opened *modally*. When a program opens a modal dialogue, the user is forced to completely handle the dialogue before any other operation can occur. When a modal dialogue is open, other modal dialogues can also be opened, but these must also be completely handled. In this way a stack of modal dialogues can be created. An example of such a situation is the behaviour of the platform dependent output file selector dialogue (which is opened by the `StdFileSelect` function `selectOutputFile`), see Figure 6.7. If the user selects an existing file, another modal dialogue is opened that asks the user if it is alright to overwrite that file. The user must answer this dialogue before the output file selector dialogue can be closed.

The type constructor class `Dialogs` contains an additional function to create dialogues in a modal way:

```haskell
class Dialogs wdef where
  openDialog :: .ls !(wdef .ls (PSt .1)) !(PSt .1)
               -> ( !ErrorReport, !PSt .1)
  openModalDialog :: .ls !(wdef .ls (PSt .1)) !(PSt .1)
```

6.8 Modal dialogues
The arguments of `openModalDialog` and `openDialog` are identical: the dialogue has a local state of type `.ls` and it has some definition of type (\texttt{def .ls (PSt .1)}) Both functions return an `ErrorReport` to inform the program if the dialogue could be created. The major difference between these two class member functions is that `openDialog` opens its argument dialogue and terminates immediately. The function `openModalDialog` opens its argument dialogue and terminates only when it is closed. This situation can only be reached by applying `closeWindow` or `closeActiveWindow` (\texttt{StdWindow}, Appendix A.47) or by terminating the parent interactive process using `closeProcess` (\texttt{StdProcess}, Appendix A.27). After the modal dialogue has been closed, its final local state value is returned. This provides a comfortable way of passing additional information from the closed modal dialogue to the calling function, as we will see in the next section.

### 6.8.1 Example: a notice extension

In this example we show the use of modal dialogues by making new `Dialogs` class instances for notices. A notice is a very simple dialogue, containing only some line(s) of text, and a (number of) button(s) that close the notice. Notices can be used by programs to briefly inform the user about some situation. The confirmation dialogue in Figure 6.7 is an example of such a notice. We will first make the notice module which contains the new `Dialogs` instance in Section 6.8.1. Using this new interactive object we can rephrase the "Hello world" program (Section 2.4). This is done in Section 6.8.1.

#### The notice module

We want to make a special instance of the `Dialogs` type constructor class that specifies a notice. As said, a notice is a dialogue that contains some line(s) of text, and a (number of) button(s). As done for every object I/O element, a notice will be defined by means of an algebraic data type:

```plaintext
:: Notice ls pst
   = Notice [String] (NoticeButton *(ls,pst)) [NoticeButton *(ls,pst)]
```
:: NoticeButton st
   = NoticeButton String (IdFun st)

The first argument of a Notice is a list of text lines. These will be shown below each other. By using one additional NoticeButton argument we can make sure that a notice always consists of at least one notice button. This mandatory notice button is placed to the right of the notice. The notice buttons will appear from right to left. A notice button is defined by NoticeButton. It is parameterised with the title and a program defined function. This function is applied after the button has been pressed and the notice has been closed.

The Notice type constructor will become an instance of the Dialogs type constructor class. This would actually be sufficient to allow a program to create both non modal and modal notices, using the class member functions openDialog and openModalDialog. To simplify creation of notices even further, we want to make the following additional function with a simpler type:

openNotice :: (Notice .ls (PSt .l)) (PSt .l) -> PSt .l

This specification is bundled in the notice definition module, which now looks as follows:

definition module notice
   // ********************************************************************
   // Clean tutorial example program.
   // //
   // // This program defines a new instance of the Dialogs class to create notices.
   // ********************************************************************

import StdWindow

:: Notice ls pst
   = Notice [String] (NoticeButton *(ls,pst)) [NoticeButton *(ls,pst)]
:: NoticeButton st
   = NoticeButton String (IdFun st)

instance Dialogs Notice

openNotice :: (Notice .ls (PSt .l)) (PSt .l) -> PSt .l

Let's work out the notice implementation module. Because a notice is a specialised dialogue, it is sufficient to map a notice definition to a dialogue definition. This mapping is done by the function noticeToDialog. For programming convenience, we will convert the text lines into text controls, and the notice buttons into button controls. Although controls are not properly introduced yet, we hope this does not frustrate your attempt to understand what's going on here.

Each notice text line is mapped to a text control. A text control is parameterised with a string, and can have a position attribute. All texts will be left-aligned. This is expressed by setting the (ControlPos (Left,zero)) attribute. For any list of strings texts this mapping can be conveniently expressed via a list comprehension (the ListLS data constructor is required for reasons explained in Chapter 7):

ListLS [TextControl text [ControlPos (Left,zero)] \ text<-texts]

We want to place the lines together in a box without additional item space (the default item space and margin values of a dialogue are platform dependent). To
make sure that we get our preferred values we override the dialogue item space and margin attributes with our own (using zero margins and an item space of three in both directions). This is done via a LayoutControl. The final mapping of the text lines results in:

texts' = LayoutControl
   ( ListLS [ TextControl text [ControlPos (Left,zero)]
      \ \ text<-texts
   ]
) [ControlHMargin 0 0,ControlVMargin 0 0,ControlItemSpace 3 3]

Let's first introduce a function noticebutton that maps one notice button to a button control. When a notice button is selected, it should first close the notice (using closeActiveWindow), and then apply the argument function. This action is handled by the argument function of the ControlFunction attribute. We allow other attributes such as placement and so on to be appended to the button control by passing it as an additional argument of noticebutton.

noticebutton (NoticeButton text f) atts
   = ButtonControl text [ControlFunction f':atts]
   where
      f' (lst,pst) = f (lst,closeActiveWindow pst)

Using noticebutton we can map the notice buttons to button controls. The mandatory notice button is mapped to the "confirm" control using the window attribute WindowOk (Section 6.2.1). We assume that okid refers to this Id. In addition, the mandatory button is right-aligned in the dialogue. So, if ok is the mandatory notice button, then it is mapped to the following button control:

ok' = noticebutton ok [ControlPos (Right,zero),ControlId okid]

The optional notice buttons appear to the left of the previous notice button. Again, we can use a list comprehension to do this mapping for a list of notice buttons buttons:

buttons' = ListLS
   [ noticebutton button [ControlPos (LeftOfPrev,zero)]
      \ \ button<-buttons
   ]

Given a Notice value (Notice texts ok buttons), noticeToDialog creates a titleless Dialog and glues the mapped notice controls using the expression (texts' :+ ok' :+ buttons')(gluing controls is also discussed in the next chapter):

noticeToDialog okid (Notice texts ok buttons)
   = Dialog "" (texts':+ok':+buttons') [WindowOk okid]

Given the mapping function noticeToDialog it is now a trivial task to define the new instance declaration of the type constructor class Dialogs:

instance Dialogs Notice where
openDialog ls notice pst
  # (okId,pst) = accPIO openId pst
  = openDialog ls (noticeToDialog okId notice) pst
openModalDialog ls notice pst
  # (okId,pst) = accPIO openId pst
  = openModalDialog ls (noticeToDialog okId notice) pst
getDialogType -
  = "Notice"

Also the definition of the convenience function openNotice now becomes trivial:

openNotice :: (Notice .ls (PSt .l)) (PSt .l) -> PSt .l
openNotice notice pst
  = snd (openModalDialog undef notice pst)

For completeness, the notice implementation module is given here.

implementation module notice

//************************************************************
// Clean tutorial example program.
// This program defines a new instance of the Dialogs class to create notices.
//************************************************************

import StdTuple, StdMisc, StdFunc
import StdId, StdPSt, StdWindow

:: Notice ls pst
  = Notice [String] (NoticeButton *(ls,pst)) [NoticeButton *(ls,pst)]
:: NoticeButton st
  = NoticeButton String (IdFun st)

instance Dialogs Notice where
openDialog ls notice pst
  # (okId,pst) = openId pst
  = openDialog ls (noticeToDialog okId notice) pst
openModalDialog ls notice pst
  # (okId,pst) = openId pst
  = openModalDialog ls (noticeToDialog okId notice) pst
getDialogType -
  = "Notice"

openNotice :: (Notice .ls (PSt .l)) (PSt .l) -> PSt .l
openNotice notice pst
  = snd (openModalDialog undef notice pst)

noticeToDialog :: Id !(Notice .ls (PSt .l))
  -> Dialog (:+: (LayoutControl (ListLS TextControl))
  (:+: ButtonControl (ListLS ButtonControl))
  ) .ls (PSt .l)
noticeToDialog okid (Notice texts ok buttons)
  = Dialog "" (texts'::+:ok'::+:buttons') [WindowOk okid]
where
texts' = LayoutControl
  ( ListLS [ TextControl text [ControlPos (Left,zero)]
  \ text<texts ]
  ) [ControlHMargin 0 0,ControlVMargin 0 0,ControlItemSpace 3 3]
ok' = noticebutton ok [ControlPos (Right,zero),ControlId okid]
buttons' = ListLS
Hello world revisited

We can now use our new notice instance to make another implementation of the “Hello world” program. It is very similar to the original version in Section 2.4 but instead of a Dialog it now uses a Notice. Figure 6.8 shows the notice. The program code is given below.

module usenotice

// Clean tutorial example program.
// This program shows "Hello world!" using a Notice.

import StdEnv, StdIO
import notice

Start :: *World -> *World
Start world
  = startIO NDI Void (openNotice hello) nil world
where
  hello = Notice ["Hello world!"] (NoticeButton "Quit" (noLS closeProcess)) nil
Chapter 7

Control handling

In the previous chapter windows and dialogues have been introduced. These top level interface elements can contain sets of controls, which are the subject of this chapter. Controls can be hierarchical, i.e. they can be composed of controls themselves. Using controls helps a program to provide a consistent and structured user interface. There are a lot of issues involved when working with controls.

First of all we introduce each of the standard controls in Section 7.1. Then the glue is introduced to build larger control structures in Section 7.2. An important aspect of controls is to manage their layout, presented in Section 7.3. Related to layout is what should happen in case a window containing controls is resized. This is discussed in Section 7.4. Finally, Section 7.5 contains a number of examples that demonstrate the use of controls.

7.1 The standard controls

The data type definitions of the standard set of object I/O library controls is given in module StdControlDef (Appendix A.7). They can be divided into three groups:

Platform standard controls
These are the controls that exist on all platforms and that have a well-defined look and feel that is platform defined. Adding these elements to your application will give your program a standardised interface that experienced users feel comfortable with.

<table>
<thead>
<tr>
<th>Control object</th>
<th>What does it look like</th>
</tr>
</thead>
<tbody>
<tr>
<td>ButtonControl</td>
<td>Button</td>
</tr>
<tr>
<td>CheckControl</td>
<td>✓ Check  ✓ Check</td>
</tr>
<tr>
<td>EditControl</td>
<td>Just text</td>
</tr>
<tr>
<td>PopUpControl</td>
<td>PopUpItem</td>
</tr>
<tr>
<td>RadioControl</td>
<td>☐ Radio ☐ Radio</td>
</tr>
<tr>
<td>SliderControl</td>
<td></td>
</tr>
<tr>
<td>TextControl</td>
<td>Just text</td>
</tr>
</tbody>
</table>

Customised controls
Although the use of platform standard controls increases the familiarity of
your application, sometimes you need to create controls with additional look and feel. For this purpose the library provides two customised controls. The look of the `CustomButtonControl` is program defined, but the library handles its feel as if it were a button. The `CustomControl` can be used when the program wants to define every aspect of its look and feel.

**Hierarchical controls**

One simple way of combining controls is to place them in a hierarchical control. Hierarchical controls always introduce a new layout scope. This functionality is provided by the `LayoutControl`. The `CompoundControl` is very analogous to a window in a control: it can have scrollbars, it has a `Picture` for its background, and so on.

Controls have an extensive set of attributes as one can see in the table below. The hierarchical controls have a number of special attributes. They are strongly related to window and dialogue attributes.

<table>
<thead>
<tr>
<th>General control attributes</th>
<th>Hierarchical control attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlActivate</td>
<td>ControlHMargin</td>
</tr>
<tr>
<td>ControlDeactivate</td>
<td>ControlHScroll</td>
</tr>
<tr>
<td>ControlFunction</td>
<td>ControlItemSpace</td>
</tr>
<tr>
<td>ControlHide</td>
<td>ControlLook</td>
</tr>
<tr>
<td>ControlId</td>
<td>ControlOrigin</td>
</tr>
<tr>
<td>ControlKeyboard</td>
<td>ControlOuterSize</td>
</tr>
<tr>
<td>ControlMinimumSize</td>
<td>ControlViewDomain</td>
</tr>
<tr>
<td>ControlModeFunction</td>
<td>ControlViewSize</td>
</tr>
<tr>
<td>ControlMouse</td>
<td>ControlVMargin</td>
</tr>
<tr>
<td>ControlPen</td>
<td>ControlVScroll</td>
</tr>
<tr>
<td>ControlPos</td>
<td></td>
</tr>
<tr>
<td>ControlResize</td>
<td></td>
</tr>
<tr>
<td>ControlSelectState</td>
<td></td>
</tr>
<tr>
<td>ControlTip</td>
<td></td>
</tr>
<tr>
<td>ControlWidth</td>
<td></td>
</tr>
</tbody>
</table>

We defer the explanation of the hierarchical control attributes until Section 7.1.11. Let’s have a look at the attributes in the left column of the table.

**ControlActivate and ControlDeactivate**

In every interactive program that has windows and dialogues, there is one window or dialogue that receives the user input, the active window. If this window or dialogue contains controls, then the user input is further directed to that particular control that has the *input focus*. When a control obtains the input focus, the function associated with its `ControlActivate` attribute is evaluated. When a control loses the input focus, the same thing happens with its `ControlDeactivate` attribute function.

**ControlFunction and ControlModeFunction**

These two attributes are the primary callback function attributes of controls. The first function is evaluated whenever the associated control is selected by the user. The same happens for the second function, except that it is also provided with the modifier keys that have been pressed at the moment the user selected the control. If these two attributes both appear in an attribute list, then the first of the two is selected.
ControlHide
This attribute defines that the control is initially invisible. It does occupy
space. By default controls are visible when element of a window or dialogue.
Of course, when some parent object is invisible, then all child controls are
invisible as well.

ControlId
This attribute identifies the control to which it is associated. If you do not
provide an Id, the control can not be modified by the program (as explained
in Chapter 4).

ControlKeyboard and ControlMouse
These attributes add keyboard and mouse handling callback functions to the
associated control. For most platform standard controls these attributes are
ignored because the platform already defines their response to these inputs.

ControlMinimumSize
This attribute defines the minimum size of the control. This value is relevant
in case of resizing (see Section 7.4). If no ControlMinimumSize is provided,
the default value zero is chosen.

ControlPen
When set, this attribute defines the initial pen attributes that the look func-
tion of customised controls uses for drawing. Pen attributes were introduced
in Section 5.1. If not present, the default pen values are used.

ControlPos
This attribute determines the layout position (see Section 7.3) of the associ-
ated control. By default, a control is placed right next to the previous control
(if it happens to be the first control, it will be positioned at the left-top).

ControlResize
This attribute defines that the control is resizeable. A control is resizeable if
it responds to user or program resize actions of a parent object. By default
controls do not resize on these events. See Section 7.4 about the resizing
behaviour of controls.

ControlSelectState
This attribute defines whether the control accepts user input (Able) or not
(Unable). Most platform standard controls give some visual clue to the user
about their current select state. By default the SelectState of controls is
Able.

ControlTip
The tip attribute provides the user of your application with a textual expla-
nation of the functionality of the control in a platform dependent way. This
explanation should be very brief (a few words only).

ControlWidth
For all platform standard controls their height is determined by their de-
manded attributes. The program can influence the width of such controls by
setting the ControlWidth attribute. The argument of this attribute is a value
of type ControlWidth which has the following type definition:

:: ControlWidth
  = PixelWidth Int
  | TextWidth String
  | ContentWidth String
The PixelWidth alternative defines the width in terms of the number of pixels. The TextWidth alternative specifies that the control should have the width exactly identical to the width of the argument string when drawn in the dialog font. If the associated control is text oriented, then the alternative ContentWidth specifies that the control should have the width as if it would contain the argument string.

In the remaining part of this section we introduce each of the standard controls. We first show the non hierarchical controls in alphabetical order, and end with the hierarchical controls. Of each control we give the set of types you must use to define it. Of the general control attributes it is indicated which are valid. This information can also be retrieved from the object I/O library. In the definition module StdControlAttribute of each control there is a predicate function that defines which attributes are valid. Finally, a simple example of a control definition is given.

7.1.1 The ButtonControl

The *button control* represents an action that should occur given the current state of the window or dialogue. The definition of a button control is as follows:

```
:: ButtonControl ls pst
 = ButtonControl String [ControlAttribute *(ls,pst)]
```

A button control has a title, given by a string. The string should not contain control characters because they will generally produce rubbish on the screen. The & character can be used to add keyboard interface to the user, depending on the platform. The escape sequence is &&. Valid control attributes are:

<table>
<thead>
<tr>
<th>ControlAttribute:</th>
<th>Valid:</th>
<th>ControlAttribute:</th>
<th>Valid:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlActivate</td>
<td></td>
<td>ControlMouse</td>
<td></td>
</tr>
<tr>
<td>ControlDeactivate</td>
<td></td>
<td>ControlPen</td>
<td></td>
</tr>
<tr>
<td>ControlFunction</td>
<td>✔</td>
<td>ControlPos</td>
<td></td>
</tr>
<tr>
<td>ControlHide</td>
<td>✔</td>
<td>ControlResize</td>
<td></td>
</tr>
<tr>
<td>ControlId</td>
<td>✔</td>
<td>ControlSelectState</td>
<td>✔</td>
</tr>
<tr>
<td>ControlKeyboard</td>
<td></td>
<td>ControlTip</td>
<td>✔</td>
</tr>
<tr>
<td>ControlMinimumSize</td>
<td></td>
<td>ControlWidth</td>
<td>✔</td>
</tr>
<tr>
<td>ControlModsFunction</td>
<td>✔</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The initial size of a button control is determined by its initial text line and ControlWidth attribute. When the button is selected by the user the first of ControlFunction and ControlModsFunction in the attribute list is evaluated. If the name of the button control is modified to a new text line, then its size is not changed.

A button control can be the confirm or cancel button of a window or dialogue (see Section 6.2.1). It must have the ControlId attribute set. The Id value must then be given to the WindowOk or WindowCancel attribute respectively.

Here is a ButtonControl with a title that contains newlines:

```
buttoncontrol
 = ButtonControl "A &Button && its text" []
```
7.1. THE STANDARD CONTROLS

7.1.2 The CheckControl

A check control is a group of check control items of which an arbitrary number can be selected. All alternatives are visible. The definition of a check control is as follows:

```plaintext
:: CheckControl ls pst
   = CheckControl [CheckControlItem *(ls,pst)] RowsOrColumns
      [ControlAttribute *(ls,pst)]
:: RowsOrColumns
   = Rows  Int
   | Columns Int
:: CheckControlItem st
:= (String,Maybe ControlWidth,MarkState,IdFun st)
```

The RowsOrColumns argument indicates the local layout of the items. Rowwise layout is given by the Rows alternative, and columnwise layout by the Columns alternative. Of each check item its title, width, initial mark state, and callback function is specified. Valid control attributes are:

<table>
<thead>
<tr>
<th>ControlAttribute</th>
<th>Valid:</th>
<th>ControlAttribute</th>
<th>Valid:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlActivate</td>
<td></td>
<td>ControlMouse</td>
<td></td>
</tr>
<tr>
<td>ControlDeactivate</td>
<td></td>
<td>ControlPen</td>
<td></td>
</tr>
<tr>
<td>ControlFunction</td>
<td></td>
<td>ControlPos</td>
<td>✓</td>
</tr>
<tr>
<td>ControlHide</td>
<td>✓</td>
<td>ControlResize</td>
<td></td>
</tr>
<tr>
<td>ControlId</td>
<td>✓</td>
<td>ControlSelectState</td>
<td>✓</td>
</tr>
<tr>
<td>ControlKeyboard</td>
<td></td>
<td>ControlTip</td>
<td>✓</td>
</tr>
<tr>
<td>ControlMinimumSize</td>
<td></td>
<td>ControlWidth</td>
<td></td>
</tr>
<tr>
<td>ControlModsFunction</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When a check control item is selected its mark state will be toggled (from Mark to NoMark and vice versa). No other check control items are affected. The corresponding callback function is then evaluated.

Here is an example of a CheckControl that consists of five items, placed in two columns. Each odd numbered item has a check mark.

```plaintext
checkcontrol
   = CheckControl
       [ ( "Check item "+toString i , Nothing
          , if (isOdd i) Mark NoMark
          , id
         )
          \ i<-[1..5]
       ] (Columns 2) []
```

7.1.3 The CustomButtonControl

A custom button control is a control that feels like a button control, but which look is customised by the program. The definition of a custom button control is as follows:

```plaintext
:: CustomButtonControl ls pst
   = CustomButtonControl Size Look [ControlAttribute *(ls,pst)]
```
Both the initial size and look of a custom button control are defined by the program. The look of a custom button control is identical to the look of a window as discussed in Section 6.4.1. The `UpdateState` argument contains zero based rectangles of the same size as the custom button control itself. Every custom button control has a *Picture environment to which the look drawing functions are applied. Valid control attributes are:

<table>
<thead>
<tr>
<th>ControlAttribute</th>
<th>Valid:</th>
<th>ControlAttribute</th>
<th>Valid:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlActivate</td>
<td></td>
<td>ControlMouse</td>
<td>✓</td>
</tr>
<tr>
<td>ControlDeactivate</td>
<td>✓</td>
<td>ControlPen</td>
<td>✓</td>
</tr>
<tr>
<td>ControlFunction</td>
<td>✓</td>
<td>ControlPos</td>
<td>✓</td>
</tr>
<tr>
<td>ControlHide</td>
<td>✓</td>
<td>ControlResize</td>
<td>✓</td>
</tr>
<tr>
<td>ControlId</td>
<td>✓</td>
<td>ControlSelectState</td>
<td>✓</td>
</tr>
<tr>
<td>ControlKeyboard</td>
<td></td>
<td>ControlTip</td>
<td></td>
</tr>
<tr>
<td>ControlMinimumSize</td>
<td>✓</td>
<td>ControlWidth</td>
<td></td>
</tr>
<tr>
<td>ControlModsFunction</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When the custom button control is selected by the user the first of `ControlFunction` and `ControlModsFunction` in its attribute list is evaluated.

A button control can be the confirm or cancel button of a window or dialogue (see Section 6.2.1). It must have the `ControlId` attribute set. The `Id` value must then be given to the `WindowOk` or `WindowCancel` attribute respectively.

The look of this `CustomButtonControl` depends on its `SelectState`. The picture on the left shows the custom button control in `Able` state, the picture on the right in `Unable` state.

```plaintext
custombuttoncontrol
  = CustomButtonControl {w=50,h=50} look []
where
  look Able {newFrame} picture
    # picture = setPenColour DarkGrey picture
    # picture = fill newFrame picture
    # picture = setPenColour Black picture
    # picture = draw newFrame picture
    = picture
  look Unable {newFrame} picture
    # picture = setPenColour LightGrey picture
    # picture = fill newFrame picture
    = picture
```

### 7.1.4 The CustomControl

A *custom control* is a control of which both the look and feel are program defined. The definition of a custom control is as follows:
:: CustomControl ls pst
= CustomControl Size Look [ControlAttribute *(ls,pst)]
:: Size = {w::Int,h::Int}
:: SelectState = Able | Unable
:: UpdateState = { oldFrame :: !ViewFrame
, newFrame :: !ViewFrame
, updArea :: !UpdateArea
}
:: ViewFrame ::= Rectangle
:: UpdateArea ::= [ViewFrame]
:: Look ::= SelectState -> UpdateState -> *Picture -> *Picture

Both the initial size and look of a custom control are defined by the program. The look of a custom control is identical to the look of a window as discussed in Section 6.4.1. The UpdateState argument contains zero based rectangles of the same size as the custom control itself. Every custom control has a *Picture environment to which the look drawing functions are applied. Valid control attributes are:

<table>
<thead>
<tr>
<th>ControlAttribute:</th>
<th>Valid:</th>
<th>ControlAttribute:</th>
<th>Valid:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlActivate</td>
<td>✓</td>
<td>ControlMouse</td>
<td>✓</td>
</tr>
<tr>
<td>ControlDeactivate</td>
<td>✓</td>
<td>ControlPen</td>
<td>✓</td>
</tr>
<tr>
<td>ControlFunction</td>
<td>✓</td>
<td>ControlPos</td>
<td>✓</td>
</tr>
<tr>
<td>ControlHide</td>
<td>✓</td>
<td>ControlResize</td>
<td>✓</td>
</tr>
<tr>
<td>ControlId</td>
<td>✓</td>
<td>ControlSelectState</td>
<td>✓</td>
</tr>
<tr>
<td>ControlKeyboard</td>
<td>✓</td>
<td>ControlTip</td>
<td>✓</td>
</tr>
<tr>
<td>ControlMinimumSize</td>
<td>✓</td>
<td>ControlWidth</td>
<td></td>
</tr>
<tr>
<td>ControlModsFunction</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The feel of a custom control is defined by its mouse and keyboard callback functions. If the user selects the custom control with the mouse, then the mouse callback function handles all input. If the custom control has the input focus, and the user is typing, then the keyboard callback function handles the keyboard input.

The look of this CustomControl depends on its SelectState. The picture on the left shows the custom control in Able state, the picture on the right in Unable state.

custombuttoncontrol
= CustomControl {w=50,h=50} look []
where
look Able {newFrame} picture
  # picture = setPenColour DarkGrey picture
  # picture = fill newFrame picture
  # picture = setPenColour Black picture
  # picture = draw newFrame picture
  = picture
look Unable {newFrame} picture
  # picture = setPenColour LightGrey picture
  # picture = fill newFrame picture
  = picture

7.1.5 The EditControl

The edit control is used to provide the user with an interface to edit (typically small amounts of) textual data. The definition of an edit control is as follows:
An edit control initially displays some text line. If the text line contains newlines then these are interpreted as line breaks. The edit control has an initial interior width (defined by the ControlWidth argument) and shows an integral number of lines. Valid control attributes are:

<table>
<thead>
<tr>
<th>ControlAttribute:</th>
<th>Valid:</th>
<th>ControlAttribute:</th>
<th>Valid:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlActivate</td>
<td>✓</td>
<td>ControlMouse</td>
<td></td>
</tr>
<tr>
<td>ControlDeactivate</td>
<td>✓</td>
<td>ControlPen</td>
<td>✓</td>
</tr>
<tr>
<td>ControlFunction</td>
<td></td>
<td>ControlPos</td>
<td>✓</td>
</tr>
<tr>
<td>ControlHide</td>
<td>✓</td>
<td>ControlResize</td>
<td>✓</td>
</tr>
<tr>
<td>ControlId</td>
<td>✓</td>
<td>ControlSelectState</td>
<td>✓</td>
</tr>
<tr>
<td>ControlKeyboard</td>
<td>✓</td>
<td>ControlTip</td>
<td>✓</td>
</tr>
<tr>
<td>ControlMinimumSize</td>
<td></td>
<td>ControlWidth</td>
<td></td>
</tr>
<tr>
<td>ControlModsFunction</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If the SelectState of an edit control is Unable, the user can not type in text. The program can keep track of the inserted text by setting the ControlKeyboard attribute. For each typed key, the keyboard function is evaluated. If the edit control has the ControlId attribute set, then its content can be retrieved using the getControlText function of module StdControl (Appendix A.4).

Here is an example of an EditControl with interior width 80 pixels and a height of three text lines.

```plaintext
editcontrol = EditControl "This is an EditControl" (PixelWidth 80) 3 []
```

### 7.1.6 The PopUpControl

A pop up control is a group of pop up control items of which exactly one item is selected. The items of a pop up control are presented in a pop up menu. Usually only the currently selected item is displayed. For this reason pop up controls consume much less space than the functionally equivalent radio controls (discussed in Section 7.1.7). The definition of a pop up control is as follows:

```plaintext
:: PopUpControl ls pst
    = PopUpControl [PopUpControlItem *(ls,pst)] Index
        [ControlAttribute *(ls,pst)]
:: PopUpControlItem st := (String,IdFun st)
:: IdFun st := st -> st
```

The initially selected item is indicated by the Index value. As a convention in the object I/O library, when indicating elements indices range from 1 up to the number of elements. So n elements are indexed by 1...n. In case the index is out of range, i.e. less than 1 or larger than n, it is set to 1 and n respectively. Valid control attributes are:
### 7.1. THE STANDARD CONTROLS

<table>
<thead>
<tr>
<th>ControlAttribute:</th>
<th>Valid:</th>
<th>ControlAttribute:</th>
<th>Valid:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlActivate</td>
<td>✓</td>
<td>ControlMouse</td>
<td>✓</td>
</tr>
<tr>
<td>ControlDeactivate</td>
<td>✓</td>
<td>ControlPen</td>
<td>✓</td>
</tr>
<tr>
<td>ControlFunction</td>
<td>✓</td>
<td>ControlPos</td>
<td>✓</td>
</tr>
<tr>
<td>ControlIdle</td>
<td>✓</td>
<td>ControlResize</td>
<td>✓</td>
</tr>
<tr>
<td>ControlKeyboard</td>
<td>✓</td>
<td>ControlSelectState</td>
<td>✓</td>
</tr>
<tr>
<td>ControlHide</td>
<td>✓</td>
<td>ControlTip</td>
<td>✓</td>
</tr>
<tr>
<td>ControlMinimumSize</td>
<td>✓</td>
<td>ControlWidth</td>
<td>✓</td>
</tr>
<tr>
<td>ControlModsFunction</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When a pop up control item is selected the previously selected item is unchecked and the new item becomes the selected item. The corresponding callback function is then evaluated. The callback function is also evaluated if the currently selected item is selected.

Here is an example of a *popUpControl* that consists of five items. The first item is the initially selected item.

```haskell
popupcontrol = PopUpControl
  [ ("PopUpItem "++toString i,id) ]
      \ i<[1..5]
    ] 1
```

#### 7.1.7 The RadioControl

A *radio control* is a group of radio control items of which exactly one item is selected. All alternatives are visible. The radio control is functionally equivalent with the pop up control (Section 7.1.6), but it consumes more space. The definition of a radio control is almost identical to that of a check control (Section 7.1.2):

```haskell
:: RadioControl ls pst
  = RadioControl [RadioControlItem *(ls,pst)] RowsOrColumns Index
                   [ControlAttribute *(ls,pst)]
:: RowsOrColumns
  = Rows    Int
      | Columns Int
:: RadioControlItem st
  ::= (String,Maybe ControlWidth,IdFun st)
:: IdFun st ::= st -> st
:: Index     ::= Int
```

The *RowsOrColumns* argument indicates the local layout of the items. Rowwise layout is given by the *Rows* alternative, and columnwise layout by the *Columns* alternative. The initially selected item is indicated by the *Index* value. As a convention in the object I/O library, when indicating elements indices range from 1 upto the number of elements. So *n* elements are indexed by *1..n*. In case the index is out of range, i.e. less than 1 or larger than *n*, it is set to 1 and *n* respectively. Of each radio item its title, width, and callback function is specified. Valid control attributes are:
When a radio control item is selected the previously selected radio control item will be unselected, and the new radio control item gets the selection mark. The corresponding callback function is then evaluated. The callback function is also evaluated when choosing the currently selected item.

Here is an example of a RadioControl that consists of five items, placed in two rows. The first item is initially selected.

```haskell
radiocontrol = RadioControl
  [ ("RadioItem"+++toString i
    ,Nothing
    ,id
    )
    \ i<-[1..5]
  ] (Rows 2) 1 []
```

### 7.1.8 The SliderControl

Slider controls are used to select a value within a range of values. The definition of a slider control is as follows:

```haskell
:: SliderControl ls pst
  = SliderControl Direction ControlWidth SliderState
      (SliderAction *(ls,pst))
    [ControlAttribute *(ls,pst)]
:: Direction   = Horizontal | Vertical
:: SliderState = { sliderMin :: !Int
                    , sliderMax :: !Int
                    , sliderThumb :: !Int
                  }
:: SliderAction st ::= SliderMove -> st -> st
:: SliderMove   = SliderIncSmall | SliderDecSmall
                  | SliderIncLarge | SliderDecLarge
                  | SliderThumb Int
```

A slider control can be given a horizontal direction (the `Horizontal` alternative of `Direction`) or a vertical direction (the `Vertical` alternative of `Direction`). In this direction it can have a certain length, which is given by the `ControlWidth` attribute.

The value range of a slider control is defined by the `SliderState` record. The initial slider state determines the integer range: `sliderMin` gives the minimum value, `sliderMax` gives the maximum value. In case these values are given in the wrong order, they will be ordered properly. The initially chosen value is given by
the `sliderThumb` value. This value must be inclusively between `sliderMin` and `sliderMax`. If a value smaller than the minimum range is given, then it is set to the minimum. If a value larger than the maximum range is given, then it is set to the maximum.

Valid control attributes for the `SliderControl` are:

<table>
<thead>
<tr>
<th>ControlAttribute</th>
<th>Valid:</th>
<th>ControlAttribute</th>
<th>Valid:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlActivate</td>
<td></td>
<td>ControlMouse</td>
<td>✓</td>
</tr>
<tr>
<td>ControlDeactivate</td>
<td>✓</td>
<td>ControlPen</td>
<td>✓</td>
</tr>
<tr>
<td>ControlFunction</td>
<td></td>
<td>ControlPos</td>
<td>✓</td>
</tr>
<tr>
<td>ControlHide</td>
<td>✓</td>
<td>ControlResize</td>
<td>✓</td>
</tr>
<tr>
<td>ControlId</td>
<td></td>
<td>ControlSelectState</td>
<td>✓</td>
</tr>
<tr>
<td>ControlKeyboard</td>
<td></td>
<td>ControlTip</td>
<td>✓</td>
</tr>
<tr>
<td>ControlMinimumSize</td>
<td></td>
<td>ControlWidth</td>
<td>✓</td>
</tr>
<tr>
<td>ControlModsFunction</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A slider control typically has five regions that can be selected by the user. These regions are shown in Figure 7.1

![Figure 7.1: The regions of the SliderControl.](image)

When the user is working with the slider control, its callback function is evaluated. The algebraic data type `SliderMove` has an alternative constructor for each of the regions of the slider control:

- `SliderDecSmall`  
  decrement arrow
- `SliderIncSmall`  
  increment arrow
- `SliderDecLarge`  
  page down region
- `SliderIncLarge`  
  page up region
- `SliderThumb`     
  thumb move
The program can decide what to do with this information. It is the responsibility of the programmer that the application responds the way the user expects.

Here is an example of a SliderControl that is oriented horizontally and has a length of 200 pixels. It results in the slider control shown in Figure 7.1.

```
slidercontrol
    = SliderControl Horizontal 200
        {sliderMin=-100,sliderMax=100,sliderThumb=0}
        (\_ st->st) []
```

### 7.1.9 The TextControl

A text control displays one line of text that can not be changed by the user. The definition of a text control is as follows:

```
:: TextControl ls pst
    = TextControl String [ControlAttribute *(ls,pst)]
```

Valid control attributes are:

<table>
<thead>
<tr>
<th>ControlAttribute</th>
<th>Valid:</th>
<th>ControlAttribute</th>
<th>Valid:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ControlActivate</td>
<td></td>
<td>ControlMouse</td>
<td></td>
</tr>
<tr>
<td>ControlDeactivate</td>
<td></td>
<td>ControlPen</td>
<td></td>
</tr>
<tr>
<td>ControlFunction</td>
<td></td>
<td>ControlPos</td>
<td></td>
</tr>
<tr>
<td>ControlHide</td>
<td>✓</td>
<td>ControlResize</td>
<td></td>
</tr>
<tr>
<td>ControlId</td>
<td>✓</td>
<td>ControlSelectState</td>
<td></td>
</tr>
<tr>
<td>ControlKeyboard</td>
<td></td>
<td>ControlTip</td>
<td>✓</td>
</tr>
<tr>
<td>ControlMinimumSize</td>
<td></td>
<td>ControlWidth</td>
<td>✓</td>
</tr>
<tr>
<td>ControlModsFunction</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The initial size of a text control is determined by its initial text line and its ControlWidth attribute. Text controls are not resizeable, and they will also not change in size in case their text line is modified by the program.

Here is an example of a text control.

```
textcontrol
    = TextControl "This is a TextControl" []
```

### 7.1.10 The LayoutControl

A layout control is a control that contains other controls. It introduces a new layout scope: i.e. controls inside it are positioned relative to the bounds of the layout control. The definition of a layout control is as follows:

```
:: LayoutControl c ls pst
    = LayoutControl (c ls pst) [ControlAttribute *(ls,pst)]
```

```
instance Controls (LayoutControl c) | Controls c
```

As you can see from the type definition, the type variable c appears on a type constructor position. The instance declaration of LayoutControl of the Controls type constructor class adds the restriction that c must be a Controls instance itself.

Valid control attributes are:
### ControlAttribute: | Valid: | ControlAttribute: | Valid:
---|---|---|---
ControlActivate |  | ControlOrigin | ✓
ControlDeactivate |  | ControlOuterSize | ✓
ControlFunction |  | ControlPen | ✓
ControlHide | ✓ | ControlPos | ✓
ControlHMargin | ✓ | ControlResize | ✓
ControlHScroll |  | ControlSelectState | ✓
ControlId | ✓ | ControlTip | ✓
ControlItemSpace | ✓ | ControlViewDomain | ✓
ControlKeyboard |  | ControlViewSize | ✓
ControlLook |  | ControlViewSize | ✓
ControlMinimumSize | ✓ | ControlVMargin | ✓
ControlModFunction |  | ControlVScroll | ✓
ControlMouse |  | ControlWidth | ✓

The size of a layout control, if not provided as a `ControlOuterSize` or `ControlViewSize` attribute, is derived by the system from its control elements (see Section 7.3 for more information on the layout of controls). Related to its size and element layout are a number of attributes. The `ControlMinimumSize` attribute determines the minimum size of the layout control (see resizing controls, Section 7.4), the `ControlResize` attribute controls the resize behaviour (see also Section 7.4), the `ControlItemSpace`, `ControlHMargin`, and `ControlVMargin` attributes define the distance between the elements themselves and the horizontal and vertical distance of the elements to the border of the layout control respectively. If a layout control does not specify any of these attributes, it obtains the same attribute values as its parent object.

### 7.1.11 The CompoundControl

The *compound control* is a control that contains other controls. It introduces a new *layout scope* (see the `LayoutControl` in Section 7.1.10). The type definition of a compound control is almost identical to that of a `LayoutControl`:

```latex
:: CompoundControl c ls pst
   = CompoundControl (c ls pst) [ControlAttribute *(ls,pst)]

instance Controls (CompoundControl c) | Controls c
```

Despite the equivalence of their type definitions, a compound control provides programmers with a lot more functionality than its cousin layout control. This becomes clear when you have a look at its valid control attributes:
The size of a compound control, if not provided as a ControlOuterSize or ControlViewSize attribute, is derived by the system from its control elements (see Section 7.3 for more information on the layout of controls). Related to its size and element layout are a number of attributes. The ControlMinimumSize attribute determines the minimum size of the compound control (see resizing controls, Section 7.4), the ControlResize attribute controls the resize behaviour (see also Section 7.4), the ControlItemSpace, ControlHMargin, and ControlVMargin attributes define the distance between the elements themselves and the horizontal and vertical distance of the elements to the border of the compound control respectively. If a compound control does not specify any of these attributes, it obtains the same attribute values as its parent object.

The compound control anatomy is the same as that of a window (Section 6.1.1). It consists of the same three layers as a window except that we call its top layer the compound frame rather than window frame. The compound frame has no title nor features like resize controls and so on.

Analogous to windows, compound controls have a view domain. As explained in Section 6.1.1, a view domain defines a finite area in which can be drawn (Chapter 5), it can also be used as an area to place controls (Section 7.3). Also for compound controls scrolling attributes can be added: ControlHScroll and ControlVScroll. These attributes control the current view frame orientation of the compound control. The left top point of the view frame that is currently visible is called the origin. This value can be set initially with the ControlOrigin attribute. The ControlLook attribute is used by the object I/O library to render the document layer of the compound control in exactly the same way as is being done for windows.

The feel of a compound control is of course partially determined by its element controls. If the ControlMouse (ControlKeyboard) attribute is given, then the compound control can handle all mouse (keyboard) events that are directed to it.

### 7.2 Control glue

In the previous section the standard set of controls has been discussed. This list does not cover all controls class instances. In the library module StdControlClass (Appendix A.6) a number of additional instances are defined, namely the type constructors :+:, ListLS,NilLS, and AddLS, NewLS (their definition can be found in module StdIOBasic, Appendix A.13). These additional instances are required to glue controls. They are treated below.
7.2 CONTROL GLUE

7.2.1 ++:

The most common constructor to glue controls is ++. Its type constructor definition and Controls class instance declaration are as follows:

```
:: ++: t1 t2 ls cs
  = (;++:) infixr 9 (t1 ls cs) (t2 ls cs)
```

```
instance Controls ((;++:) c1 c2) | Controls c1 & Controls c2
```

Assume we have two Controls instances c1 and c2, working on the same local state of type ls and context state of type cs. Now the glued expression c1++c2 is also a Controls instance working on the same local state and context state. Because ++ is right associative, the expression c1++c2++c3 should be read as c1++(c2++c3).

As an example, reconsider the button control definition buttoncontrol in Section 7.1.1, and the edit control definition editcontrol in Section 7.1.5:

```
buttoncontrol
  = ButtonControl "A &Button &nbsp its text" []
editcontrol
  = EditControl "This is an EditControl" (PixelWidth 80) 3 []
```

Now the following expressions are legal combinations: (buttoncontrol ++: editcontrol) and (editcontrol ++: buttoncontrol).

7.2.2 ListLS and NilLS

The ++ type constructor is not always the most appropriate glue. When constructing sets of controls of the same type, it is much more convenient to use lists and list comprehensions. When constructing an unknown number of controls, it is even impossible to use ++. Again, lists provide more flexibility. List-like glue is provided by the type constructor ListLS. The type constructor NilLS is a shorthand for ListLS []. It can also be conveniently used to state that a hierarchical control, window, or dialogue has no controls. Here are the definitions:

```
:: ListLS t ls cs = ListLS [t ls cs]
:: NilLS    ls cs = NilLS
```

```
instance Controls (ListLS c) | Controls c
instance Controls NilLS
```

Given a list of Controls instances xs = [c1...cn], working on the same local state of type ls and context state cs, the expression ListLS xs is also a Controls instance working on the same local state and context state.

As an example, assume that you have a list of text labels, labels, and that you want to create a set of controls in the following way: for each label in labels, create a text control with content that label, and right to that text control an empty edit control that has a width of 100 pixels. The edit controls have to be positioned at the same x-coordinate. The method has to work for any list length and of course handle labels of different length. One additional nasty property is that text controls and edit controls can have different heights (platform dependent).

Here's how to do it. First you determine the longest label size:
maxlength = maxLength (map size labels)

Given this value, you select the longest label from labels:

maxlabel = hd (filter (\l->size l==maxlength) labels)

Now you can use ListLS to create an arbitrary number of text controls glued with edit controls, using a list comprehension:

ListLS
[ TextControl label [ControlWidth (ContentWidth maxlabel)
   ,ControlPos (Left,zero)
   ]
   ,:+
   EditControl "" (PixelWidth 100) 1 []
 \ label<-labels
 ]

When applied to the following value of labels:

labels = [ "A short label"
   ,"A fairly long and verbose label"
   ,"Another lengthy label"
   ]

and put it in a dialogue, the result is as in Figure 7.2.

![Figure 7.2: Glueing with list comprehensions.](image)

### 7.2.3 AddLS and NewLS

The previously discussed glueing type constructors always glue controls that work on the same local state and context state. Two other glueing constructors are defined to extend and change the local state, AddLS and NewLS.

Given a Controls instance c1 that works on a local state of type ls and a context state of type cs, one can add another Controls instance c2 that works on an extended local state of type (new,ls) and the same context state of type cs. Let x be a value of type new, then this is done by the expression c1 :+: {addLS=x, addDef=c2}.

:: AddLS t ls cs = E . new: { addLS::new, addDef::t *(new,ls) cs }

instance Controls (AddLS c) | Controls c
Given a `Controls` instance `c1` that works on a local state of type `ls` and a context state of type `cs`, one can add another `Controls` instance `c2` that works on a `new` local state of type `new` and the same context state of type `cs`. Let `x` be a value of type `new`, then this is done by the expression `c1 :+: {newLS=x, newDef=c2}`.

```
:: NewLS t ls cs = E. .new: { newLS::new, newDef::t new cs }
```

```
instance Controls (NewLS c) | Controls c
```

In both cases the extended part of the local state and the new local state are encapsulated completely from the external context using existential quantification.

As an example of gluing controls in this way, let’s implement a manually incrementable counter control. To do this we fill a `LayoutControl` with three other controls. To display the current count value an `Unable EditControl` is used. Two `ButtonControls` are used to decrement and increment the counter.

```
counter displayid
  = {newLS =count0
      ,newDef=LayoutControl
        ( EditControl (toString count0) (ContentWidth "000") 1  
          [ControlSelectState Unable  
            ,ControlPos (Left,zero)  
            ,ControlId displayid  
          ]  
        :+: ButtonControl "-" [ControlFunction (count (-1))]  
        :+: ButtonControl "+" [ControlFunction (count 1)]  
        )  
        [ControlHMargin 0 0  
         ,ControlVMargin 0 0  
         ,ControlItemSpace 0 0  
        ]
  }
where
  count0 = 0

  count :: Int (Int,PSt .1) -> (Int,PSt .1)
  count dx (count,pst=:{io})
    = ( count+dx  
        , appPIO (setControlText displayid (toString (count+dx))) pst  
    )
```

Figure 7.3 shows the counter after some user manipulations that resulted in the counter value -15.

![Counter Control](image)

Figure 7.3: The counter control.

### 7.3 Control layout

The object I/O library offers the programmer an expressive layout mechanism to define the layout of controls. As we have seen before, controls can be element
of windows, dialogues, layout and compound controls. The layout rules that are discussed in this section apply to each of these cases. Some layout rules refer to the view domain and view frame of the parent object. In case of dialogues and layout controls these two are identical: they are zero based rectangles with a size equal to the interior size of the parent object.

As we have seen, every control can have a ControlPos attribute. This attribute is defined as follows:

```plaintext
:: ControlAttribute st
  = ... | ControlPos ItemPos | ...
:: ItemPos
  ::= (ItemLoc,ItemOffset)
:: ItemLoc
  = Fix
  | LeftTop   | RightTop   | LeftBottom | RightBottom
  | Left      | Center     | Right
  | LeftOf Id | RightTo Id | Above Id   | Below Id
  | LeftOfPrev| RightToPrev| AbovePrev  | BelowPrev
:: ItemOffset
  = NoOffset
  | OffsetVector Vector2
  | OffsetFun ParentIndex OffsetFun
:: ParentIndex
  ::= Int
:: OffsetFun
  ::= (ViewDomain,Point2) -> Vector2
```

The layout position of a control consists of two values: an ItemLoc value and an ItemOffset value. The ItemLoc actually determines the location of the control, the ItemOffset value adds an offset to this position (which of course influences the position of other controls). The ItemLoc values can be divided into four groups:

**Fixed position**
This is only the Fix alternative of ItemLoc. Fix places the left top corner of the associated control at the left top value of the view domain of its parent object. The ItemOffset determines the actual position in the view domain, and is given in the coordinate system of the view domain of the parent object.

You can use this layout attribute when you want to ‘pin’ a control on the document layer of a parent object, rather than its control layer (see Section 6.1.1). For dialogues and layout controls, that do not have a document layer, this layout attribute is identical to the boundary aligned attribute LeftTop.

In a window or compound control, scrolling the view frame will also scroll all controls that have a fixed position.

**Boundary aligned**
These are the alternatives LeftTop, RightTop, LeftBottom and RightBottom. These attributes place their associated control at the left-top, right-top, left-bottom, and right-bottom respectively of the current view frame of the parent object.

**Line aligned**
These are the alternatives Left, Center, and Right. These attributes place
their associated control below all previous line aligned controls in the vertical position, and left-aligned, centered, or right-aligned respectively with respect to the current view frame of the parent object in the horizontal direction.

**Relative position**

These are the alternatives LeftOf, RightTo, Above, Below, and LeftOfPrev, RightToPrev, AbovePrev, and BelowPrev. The first four alternatives must be parameterised with the Id of a control that is element of the same parent object, otherwise a runtime error will occur. The latter four alternatives can be defined in terms of the first four but have the advantage that you do not have to create Id values for controls that you only want to relatively place other controls to. Placing controls relatively to other controls must construct a tree of related controls: cyclic references are not allowed and result also in a runtime error.

Controls that have a relative position layout form a layout tree with one layout root control. The layout attribute of the layout root control determines the layout positions of the whole layout tree. If it is at a fixed position, the layout tree obtains a fixed position. If it is boundary aligned, the layout tree is aligned at the same boundary. If it is line aligned, the layout tree is line aligned.

Except for the first layout root control the default layout attribute for controls is (RightToPrev, zero). For the first layout root control the default attribute is (Left, zero). Consequently, the default layout order is from left to right in one single line.

Controls are allowed to overlap partially or completely. This is particularly useful in case of combinations of hidden and visible controls when at all times only one is visible. It allows the program to change the control structure in an easy way by hiding and showing controls.

In the remaining part of this section a number of examples are given to illustrate control layout. In each of the examples we assume that the controls that are being laid out are placed in a parent object with a view domain and view frame as given in Figure 7.4. The x axis (the horizontal arrow) and y axis (the vertical arrow) intersect at the coordinate zero.

![Figure 7.4: View domain and view frame.](image)

![Figure 7.5: A layout tree of five controls.](image)
CHAPTER 7. CONTROL HANDLING

Controls are being displayed as boxes. Figure 7.5 shows the control configuration that is used in the examples. It consists of five equally sized controls, \( c_0 \ldots c_4 \). The layout root control is \( c_0 \). The controls \( c_1 \ldots c_4 \) have a relative layout to \( c_0 \) with layout attributes \texttt{LeftOf}, \texttt{RightTo}, \texttt{Above}, and \texttt{Below} respectively. The offsets are zero.

### 7.3.1 Layout at fixed position

Controls and layout trees that have a \texttt{Fix} layout attribute are being placed relative to the view domain of the parent object. So their visibility depends on the current orientation of the parent view frame (recall that the view frame clips everything that is outside of it). Figure 7.6 shows the control configuration of Figure 7.5 when the layout root control has the attribute \texttt{(Fix,zero)}.

![Figure 7.6: The layout tree at (Fix,zero).](image)

### 7.3.2 Layout at view frame boundary

Placing a control at a boundary aligned position ensures that the control is always visible, because its position is relative to the view frame of its parent object (provided that the view frame is large enough). If this control is the layout root, then the visibility of the controls in its layout tree depend on their relative position. For instance, if a layout root control is placed at the left-top of the view frame (using \texttt{LeftTop}), then relatively placed controls at its left and above it are invisible, assuming all controls have zero offsets.

![Figure 7.7: The layout tree at LeftTop, RightTop, LeftBottom, and RightBottom.](image)
This is illustrated in Figure 7.7. It shows the positions of the control configuration of Figure 7.5 when positioned at every corner of the view frame using LeftTop, RightTop, LeftBottom, and RightBottom respectively with zero offsets.

### 7.3.3 Layout in lines

Laying out controls and layout trees in lines is similar to writing characters in an English piece of text: each next control is placed right next to the previous control until a new line is started. A new line starts below the previous line. Lines can be left aligned, centered, or right aligned. The layout attributes Left, Center, and Right introduce a new line and its alignment. The first line starts at the top of the view frame. If the view frame is not large enough to hold all the lines, then the controls in these lines will not be visible.

![Layout example](image)

**Figure 7.8:** The layout tree at Left, Center, and Right.

This is illustrated in Figure 7.8. It shows the positions of the control configuration of Figure 7.5 when positioned at Left, Center, and Right respectively, using zero offsets.

### 7.3.4 Layout offsets

So far we have used zero offsets in the layout attribute examples. The layout position of a control is changed by an offset vector value $\mathbf{v} = \{vx, vy\}$ as follows: first, the layout position of the control is calculated as explained above, using a zero offset. Now assume that this results in the exact location $\mathbf{pos} = \{x,y\}$. Then the real position of the control is $\{x= x+vx, \ y= y+vy\}$. Figure 7.9 illustrates this. Given two controls $c_0$ and $c_1$ it shows the result of placing $c_1$ at RightTo control $c_0$ with an offset value $\mathbf{v} = \{vx, vy\}$. The dashed box shows the location of $c_1$ using a zero offset.

![Offset example](image)

**Figure 7.9:** Laying out controls using an offset vector.
7.3.5 Layout relative to the previous control

As explained earlier in this section, the default layout attribute of a control is (RightToPrev, zero). The other layout attributes that refer to the previous control are LeftOfPrev, AbovePrev, and BelowPrev. In this section we explain what the previous control is.

Section 7.2 introduced the glue to create control structures. The best way to look at such a control structure is to have a look at its numbered graph structure. Consider the following expression: (a++b++c) with a, b, and c standard Controls class instances as introduced in Section 7.1. Figure 7.10 shows the graph structure (recall that ++ is right associative).

![Graph Structure](image)

Figure 7.10: The numbered graph of (a++b++c).

Each node in the graph has an index. If a node is a glue node, then first number the left sub tree, then the node itself, then the right sub tree. If a node is a standard Controls class instance, then number it. The nodes of the sub tree of a CompoundControl are not numbered. Proceeding in this way, one obtains the index figures at each of the nodes in Figure 7.10. If a node in the graph with index \( i \) represents one of the standard Controls class instances then its previous control is represented by that node in the graph that has the highest index less than \( i \) and represents also one of the standard Controls class instances. So the previous control of \( c \) is not \( ++_1 \), but \( b \) because we assumed that \( b \) is an instance of the standard Controls class. Analogously, the previous element of \( b \) is neither one of the two \( ++ \) nodes, but \( a \). Finally, \( a \) has no previous control.

7.4 Resizing controls

The object I/O system has a simple mechanism to let controls respond to resize actions of their parent object. If a control wants to respond to resize events, it should have a ControlResize attribute. It is defined as follows:

:: ControlAttribute st
    = ... | ControlResize ControlResizeFunction | ...
:: ControlResizeFunction
    ::= Size -> Size -> Size -> Size

The control resize function is applied to the current outer size of the associated control, the old view frame size of its parent object, and the new view frame size of its parent object. The result size is supposed to be its new outer size. This calculation is performed for all controls that are part of the object that is being resized. If a hierarchical control has a resize function, and its new size is different from its old size, then this computation continues recursively, otherwise the layout of its elements is not recalculated. Given the new sizes of the controls, the layout is
recalculated and adjusted accordingly. The effect of this strategy is that the relative layout of controls is never changed.

As an example, consider one wants to have a CompoundControl that always displays three CustomControls next to each other at the top of its view frame. The CompoundControl takes care that its view frame width is always dividable by three, using its own ControlResize function compoundresize. Its ControlLook function draws a rectangle fitting its current view frame.

```plaintext
compound = CompoundControl
  (ListLS [custom,custom,custom])
  [ControlResize compoundresize
    ,ControlViewSize compoundsize
    ,ControlLook True (\_ {newFrame}->draw newFrame)
    ,CompoundHMargin 0 0
    ,CompoundVMargin 0 0
    ,CompoundItemSpace 0 0
  ]
compoundsize = {w=60,h=75}
compoundresize _ _ newparentszie=:{w}
  = {newparentszie & w=w/3*3}
```

The CustomControls resize their widths according to the new width of their parent control, using the ControlResize function customresize. For their look function we reuse the look function mentioned in Section 6.4.1, page 57. This function draws a rectangle fitting its current view frame and its two diagonals.

```plaintext
custom = CustomControl {w=compoundsize.w/3,h=6} crossbox
  [ControlResize customresize]
customresize customsize _ {w}
  = {customsize & w=w/3}
```

Figure 7.11 shows what happens with the controls when the parent object is resized. At the left the initial state of the CompoundControl and its CustomControls is displayed. As explained in Section 7.3, the three custom controls form a layout tree with the layout root control having the layout attribute (Left,zero) and the other CustomControls (RightToPrev,zero). In the middle, the CompoundControl is resized to the right and bottom. This resize action causes first recalculation of the size of the CompoundControl, using compoundresize. Because this value differs from the old size, recalculation continues for each CustomControl. The final result is shown at the right.

### 7.5 Examples

Here are a number of additional examples of using controls. The key spotting and mouse spotting examples already handled in the previous chapter are revisited.

#### 7.5.1 Keyspotting revisited

In this example we extend the keyspotting example in Section 6.7.1 with the possibility to monitor also the keyboard input of controls. We create a `Window` that
contains a CompoundControl that contains a CustomControl. Before we discuss each of the components below, we have a look at the way they handle keyboard input. A screenshot of the program is given in Figure 7.12.

Each of the components is keyboard sensitive and uses the same KeyboardFunction, spotting. This function is almost identical to the one presented in Section 6.7.1. The only difference is that the keyboard input is now shown in the CustomControl instead of the window. For this purpose spotting is parameterised with the Id of the CustomControl. It is also parameterised with a string that states who currently has the input focus. Now spotting changes the Look function of the CustomControl and forces its update (by using a True boolean).

spotting cid who x pst
   = appPIO (setControlLook cid True (True,look text)) pst
where
   text = who+++"+++toString x

The Look function, look, of the CustomControl is almost identical to the original look function: except that it centers its argument string it also draws a rectangle around it, so that we can easily see where the components are.

look text _ {newFrame} picture
   # picture = unfill newFrame picture
   # picture = draw newFrame picture
   # (width,picture)= getPenFontStringWidth text picture
   = drawAt {x=(w-width)/2,y=h/2} text picture
where
The CustomControl, \texttt{custom}, displays which component is currently receiving what keyboard input. The control is identified by \texttt{cid}. It parameterises its Keyboard-Function spotting with its Id and the string "Control". Its initial look draws a box around itself. The control is resizable, specified by adding the ControlResize attribute. Whenever the parent object is resized, \texttt{custom} changes its size in exactly the same amount.

\begin{verbatim}
custom = CustomControl customsize (look "")
    [ ControlKeyboard (const True) Able
        (noLS1 (spotting cid "Control"))
        , ControlId cid
        , ControlResize resize
    ]

resize oldCSize oldParentSize newParentSize
    = { w = oldCSize.w+newParentSize.w-oldParentSize.w
        , h = oldCSize.h+newParentSize.h-oldParentSize.h
    }
\end{verbatim}

The CompoundControl, \texttt{compound}, contains only \texttt{custom}. It parameterises its KeyboardFunction spotting with the Id of \texttt{custom} and the string "Compound". Its initial view frame size is chosen such that it is large enough to display \texttt{custom} completely. It conveniently uses the look function to draw a box around itself. The compound control is also resizable, and uses the same resize function as \texttt{custom}.

\begin{verbatim}
compound = CompoundControl custom
    [ ControlKeyboard (const True) Able
        (noLS1 (spotting cid "Compound"))
        , ControlViewSize {w=customsize.w+2*margin
                           ,h=customsize.h+2*margin
        }
        , ControlResize resize
        , ControlLook True (look "")
    ]
\end{verbatim}

Finally, the Window, \texttt{window}, contains only \texttt{compound}. It parameterises its spotting function with the Id of \texttt{custom} and the string "Window". Its initial size is chosen such that it is large enough to display \texttt{compound} completely. For this purpose also the margin attributes are set. Termination of the program is taken care of by having the program quit when the user closes the window.

\begin{verbatim}
window = Window "keyspotting" compound
    [ WindowKeyboard (const True) Able
        (noLS1 (spotting cid "Window"))
        , WindowViewSize {w=customsize.w+4*margin
                           ,h=customsize.h+4*margin
        }
        , WindowHMargin margin margin
        , WindowVMargin margin margin
        , WindowClose (noLS closeProcess)
    ]
\end{verbatim}
Remaining details that need to be defined are the actual creation of the interactive program and its window. For completeness, we include the program code of keyspotting below.

```haskell
module keyspotting

// *******************************************************
// Clean tutorial example program.
// This program monitors keyboard input that is sent to a Window which consists
// of a CompoundControl which consists of a CustomControl.
// *******************************************************

import StdEnv,StdIO

Start :: *world -> *World
Start world
  # (cid,world) = openId world
  # custom = CustomControl customsize (look "")
    [ ControlKeyboard (const True) Able
      (noLS1 (spotting cid "Control"))
      , ControlId cid
      , ControlResize resize
    ]

  # compound = CompoundControl custom
    [ ControlKeyboard (const True) Able
      (noLS1 (spotting cid "Compound"))
      , ControlViewSize {w=customsize.w+margin
          , h=customsize.h+margin
        }
      , ControlResize resize
      , ControlLook True (look "")
    ]

  # window = Window "keyspotting" compound
    [ WindowKeyboard (const True) Able
      (noLS1 (spotting cid "Window"))
      , WindowViewSize {w=customsize.w+margin
          , h=customsize.h+margin
        }
      , WindowHMargin margin margin
      , WindowVMargin margin margin
      , WindowClose (noLS closeProcess)
    ]
  = startIO SDI Void (snd o openWindow Void window)

where

  customsize = {w=550,h=100}
  margin = 10
  resize oldSize oldParentSize newParentSize
    = { w = oldSize.w+newParentSize.w-oldParentSize.w
        , h = oldSize.h+newParentSize.h-oldParentSize.h
    }
  spotting cid who x pst
    = appPIO (setControlLook cid True (True,look text)) pst
  where

    look text _ (newFrame) picture
      # picture = unfill newFrame picture
      # picture = draw newFrame picture
      # (width,picture) = getPenFontStringWidth text picture
      # (x,y) = redrawAt x-(w-width)/2,y-h/2 text picture

  where

  {v,h} = rectangleSize newFrame
```

7.5. **EXAMPLES**

![Example Image](image)

**Figure 7.13:** The mousespotting program in action.

### 7.5.2 Mousespotting revisited

In this example we extend the mouse spotting example of Section 6.7.2. Just like then, the revised mouse spotting example is almost identical to the revised keyspotting example discussed above. The only differences are the title of the window and the replacement of the keyboard attributes by mouse attributes. A screenshot of the program is given in Figure 7.13. For completeness, we show the code of the revised mouse spotting example below.

```plaintext
module mousespotting

// Clean tutorial example program.

// This program monitors mouse input that is sent to a Window which consists
// of a CompoundControl which consists of CustomControl.

import StdEnv, StdIO

Start :: *World -> *World

Start world
    # (cid,world) = openId world
    # custom = CustomControl customsize (look 
    [ ControlMouse (const True) Able
        (noLS1 (spotting cid "Control"))
        . ControlId cid
        . ControlResize resize
    ]
    # compound = CompoundControl custom
    [ ControlMouse (const True) Able
        (noLS1 (spotting cid "Compound"))
        . ControlViewSize {w=customsize.w+2*margin, h=customsize.h+2*margin}
        . ControlResize resize
        . ControlLook True (look 
    ]
    # window = Window "mousespotting" compound
    [ WindowMouse (const True) Able
        (noLS1 (spotting cid "Window"))
        . WindowViewSize {w=customsize.w+4*margin, h=customsize.h+4*margin}
        . WindowHMargin margin margin
        . WindowVMargin margin margin
        . WindowClose (noLS closeProcess)
    ]
```
- startIO SDI Void (snd o openWindow Void window)
  [ProcessClose closeProcess] world

where
customSize = {w=550,h=100} // start
margin = 10

resize oldCSizes oldParentSize newParentSize
  { w = oldCSizes.w+newParentSize.w-oldParentSize.w
    , h = oldCSizes.h+newParentSize.h-oldParentSize.h
  }

spotting cid who x pst
  = appPI0 (setControlLook cid True (True,look text)) pst

where
text = who++"":""+toString x

look text _ {newFrame} picture
  # picture = unfill newFrame picture
  # picture = draw newFrame picture
  # (width,picture) = getFontStringWidth text picture
  = drawAt {x=(w-width)/2,y=h/2} text picture

where
  {w,h} = rectangleSize newFrame
Chapter 8

Menus

Many interactive applications allow a user to manipulate a number of documents. As we saw in the previous chapter, the user can issue these manipulations by means of the keyboard and mouse. Another common source of manipulations is by issuing commands to the application. This is where menus come in. Menus help a program to structure the set of available commands. To the user of an application, the use of menus provides a consistent and easily browsable graphical display of the set of available commands. For these reasons it is recommended to use menus in a program.

In Section 8.1 we introduce the standard set of menu definitions that are at a programmers disposal. There are two top-level menus, the common menu, and the pop up menu. Then the glue is introduced to create larger menu structures in Section 8.2. One special menu is available for interactive processes that have the multiple document interface (MDI) attribute, the windows menu. This menu enumerates the currently open and visible windows of that process and give some commands to organise them. This is treated in Section 8.3. Menus provide a consistent graphical interface to users. To enhance the consistency, a number of programming conventions have evolved. These are discussed in Section 8.4.

8.1 Menus and menu elements

Menus and menu elements can be defined by means of the type definitions in module StdMenuDef (Appendix A.19). There are two top-level menu objects: the menu that can be found in the menu bar of an application, and the pop up menu that can be opened at any place in the application area (usually in a window). Menu objects are instances of the type constructor class Menus. This class and the instance declarations can be found in StdMenu.

class Menus mdef where
    openMenu :: .1s !(mdef .1s (PSt .1)) !(PSt .1)
    -> (!ErrorReport,'PSt .1)

instance Menus (Menu m) | MenuElements m
instance Menus (PopUpMenu m) | PopUpMenuElements m

Menus and pop up menus share most of the menu elements. The only exception is that pop up menus are not allowed to have sub menus. This restriction is enforced
by the two separate menu element classes for the two menus, MenuElements and PopUpMenuElements respectively.

Menus and their elements share the same set of menu attributes. Before we introduce each of the menu components, we will focus on the attributes first.

### 8.1.1 The menu attributes

The menu attributes are used by both menus and menu elements. They are the following:

```plaintext
:: MenuAttribute st
  = MenuId Id
  | MenuSelectState SelectState
  | MenuIndex Int
  | MenuInit (IdFun st)
  | MenuFunction (IdFun st)
  | MenuMarkState MarkState
  | MenuModsFunction (ModifiersFunction st)
  | MenuShortKey Char
```

**MenuId**

This attribute identifies the menu or menu element to which it is associated. If you do not provide a MenuId the menu (element) can not be modified by the program.

**MenuSelectState**

This attribute defines whether the menu (element) can be used by the user (Able) or not (Unable). Usually this will affect the look of the menu (element). The default value is Able.

**MenuIndex**

This attribute defines the index position of a menu. Index positions range from one (for the first menu) up to the number of menus. A negative or zero MenuIndex attribute value will place the menu in front of all current menus. A MenuIndex attribute value that is larger than the current number of menus will place the menu behind all current menus. Other MenuIndex attribute values place the menu behind the menu with that index value.

**MenuInit**

This attribute defines an action that should be performed immediately after opening the menu. This is equivalent to the window initialisation action (see Section 6.2.1). If no MenuInit attribute is provided, no additional action is performed.

**MenuFunction and MenuModsFunction**

These two attributes add callback functions to menu elements that are evaluated when the menu element to which they are associated is selected by the user. The difference between these two attributes is that the former is simply evaluated whenever the menu element is selected, and that the latter provides the callback function with the modifier keys that have been pressed at the moment of selecting the menu element. In a MenuAttribute list, the first of these two attributes is chosen.

**MenuMarkState**

This attribute adds a check mark symbol (Mark) or leaves it out (NoMark) to a menu element. The default value is NoMark.
MenuShortKey
This attribute defines a character that can be used by user to select the menu element to which the character is associated by means of the keyboard. The menu element can be selected by pressing that character and some special, platform dependent modifier key.

The object I/O library also supports the standard Windows keyboard interface of user selection of menus and menu elements. In this interface, a menu (element) can be selected by the keyboard by marking one of the characters in its title. A character is marked by prefixing it with ‘&’. The escape sequence is ‘&amp;’.

8.1.2 The Menu

A *menu* is a top level interface element that contains a group of related commands. The menu appears in the menu bar of the interactive process. The definition of a menu is as follows:

```
:: Menu m ls pst = Menu Title (m ls pst) [MenuAttribute *(ls,pst)]
```

instance Menus (Menu m) | MenuElements m

The usual appearance of a menu is by its title, which is displayed in the menu bar. The user can browse through its commands by mouse or by a platform dependent keyboard interface. Valid menu attributes are:

<table>
<thead>
<tr>
<th>MenuAttribute:</th>
<th>Valid:</th>
<th>MenuAttribute:</th>
<th>Valid:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MenuFunction</td>
<td></td>
<td>MenuMarkState</td>
<td></td>
</tr>
<tr>
<td>MenuId</td>
<td>✓</td>
<td>MenuModsFunction</td>
<td></td>
</tr>
<tr>
<td>MenuIndex</td>
<td>✓</td>
<td>MenuSelectState</td>
<td></td>
</tr>
<tr>
<td>MenuInit</td>
<td></td>
<td>MenuShortKey</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Example** The left picture shows the menu when not selected by the user, the right picture when selected by the user.

```plaintext
menu
  = Menu "File"
    ( MenuItem "Open..." [MenuShortKey 'o']
      :: MenuItem "Close" [MenuSelectState Unable,
                            MenuShortKey 'w']
    )
  :: MenuItem "Quit" [MenuShortKey 'q']
) []
```

8.1.3 The PopUpMenu

A *pop up menu* is a top level interface element that contains a group of related commands. It can appear at any place in an interactive process. Its definition is actually a restricted version of that of a menu:
:: PopUpMenu m ls pst = PopUpMenu (m ls pst)

instance Menus (PopUpMenu m) | PopUpMenuElements m

Pop up menus have no title and no attributes.

Example Here is an example of a pop up menu, containing the same elements as the menu example above. The image to the left shows the menu when opened, the image to the right when the user selects an item.

```plaintext
menu
  = PopUpMenu
    ( MenuItem "Open..." [MenuShortKey 'o']
      :+: MenuItem "Close" [MenuSelectState Unable
        ,MenuShortKey 'w'
      ]
      :+: MenuItem "Quit" [MenuShortKey 'q']
    )
```

8.1.4 The MenuItem

The *menu item* is the standard element that refers to a command. The definition of a menu item is as follows:

```plaintext
:: MenuItem ls pst = MenuItem Title [MenuAttribute *{ls,pst}]
```

instance MenuElements MenuItem
instance PopUpMenuElements MenuItem

The title of a menu element is displayed as a member of its parent menu. When the user selects the menu item its Menu(Mods)Function attribute is evaluated if the menu item, and all of its parent menus are Able. The appearance of the menu item reflects this state. Valid menu item attributes are:

<table>
<thead>
<tr>
<th>MenuAttribute</th>
<th>Valid</th>
<th>MenuAttribute</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>MenuFunction</td>
<td>✓</td>
<td>MenuMarkState</td>
<td>✓</td>
</tr>
<tr>
<td>MenuItem</td>
<td>✓</td>
<td>MenuModesFunction</td>
<td>✓</td>
</tr>
<tr>
<td>MenuIndex</td>
<td>✓</td>
<td>MenuSelectState</td>
<td>✓</td>
</tr>
<tr>
<td>MenuInit</td>
<td></td>
<td>MenuShortKey</td>
<td>✓</td>
</tr>
</tbody>
</table>

An example of a menu item is given in the previous sections 8.1.2 and 8.1.3.

8.1.5 The MenuSeparator

The *menu separator* is a menu element that is only used to provide users with visual clues that separate groups of related menu elements within a parent menu. Graphically, a menu separator usually inserts some vertical space and a horizontal divider within a menu. Menu separators have no further functionality. The definition of a menu separator is as follows:
Because menu separators have no other purpose than providing some 'white space' between menu elements, the only valid menu separator attribute is the MenuId (which can be used to remove a separator dynamically):

<table>
<thead>
<tr>
<th>MenuAttribute:</th>
<th>Valid:</th>
<th>MenuAttribute:</th>
<th>Valid:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MenuFunction</td>
<td></td>
<td>MenuMarkState</td>
<td></td>
</tr>
<tr>
<td>MenuId</td>
<td>✓</td>
<td>MenuModsFunction</td>
<td></td>
</tr>
<tr>
<td>MenuIndex</td>
<td></td>
<td>MenuSelectState</td>
<td></td>
</tr>
<tr>
<td>MenuInit</td>
<td></td>
<td>MenuShortKey</td>
<td></td>
</tr>
</tbody>
</table>

In the menu examples in the sections 8.1.2 and 8.1.3 menu separators have been used.

### 8.1.6 The RadioMenu

A radio menu element is a group of menu items of which exactly one menu item is selected. All alternatives are visible. The definition of a radio menu is as follows:

```haskell
:: RadioMenu ls pst = RadioMenu [MenuRadioItem *(ls,pst)] Index
                         [MenuAttribute *(ls,pst)]
:: MenuRadioItem st := (Title,Maybe Id,Maybe Char,IdFun st)

instance MenuElements RadioMenu
instance PopUpMenuElements RadioMenu
```

The initially selected item is indicated by the Index argument. As a convention in the object I/O library, when indicating elements indices range from 1 upto the number of elements. So n elements are indexed by 1…n. In case the index is out of range, i.e. less than 1 or larger than n, it is set to 1 and n respectively. Valid radio menu attributes are:

<table>
<thead>
<tr>
<th>MenuAttribute:</th>
<th>Valid:</th>
<th>MenuAttribute:</th>
<th>Valid:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MenuFunction</td>
<td></td>
<td>MenuMarkState</td>
<td></td>
</tr>
<tr>
<td>MenuId</td>
<td>✓</td>
<td>MenuModsFunction</td>
<td></td>
</tr>
<tr>
<td>MenuIndex</td>
<td></td>
<td>MenuSelectState</td>
<td></td>
</tr>
<tr>
<td>MenuInit</td>
<td></td>
<td>MenuShortKey</td>
<td>✓</td>
</tr>
</tbody>
</table>

When an item of the radio menu is selected the previously selected radio menu item will be unchecked, and the new radio menu item gets the check mark. The corresponding callback function is then evaluated. The callback function is also evaluated if the currently selected radio menu item is selected.

**Example** A radio menu and its initial look (it is instructive to compare this with the radio control example at page 82).

radiomenu
CHAPTER 8. MENUS

= RadioMenu
  [ ("Radio item "++toString i,Nothing,Just (iChar i),id)
    \ i<-[1..5]
  ] 1 []

where
  iChar i = toChar (toInt '1'+i-1)

8.1.7 The SubMenu

The sub menu is a menu element that contains other menu elements. So it is a menu within a menu, and can of course contain sub menus as well. The definition of a sub menu is therefore very similar to that of a menu:

::: SubMenu m ls pst
  = SubMenu Title (m ls pst) [MenuAttribute *(ls,pst)]

instance MenuElements (SubMenu m) | MenuElements m

The usual appearance of a sub menu is by its title. This title appears in the menu that contains the sub menu. On many systems some additional visual feedback is provided to inform the user that this item is a sub menu. The user can browse through its elements by mouse or by a platform dependent keyboard interface. Valid sub menu attributes are:

<table>
<thead>
<tr>
<th>MenuAttribute:</th>
<th>Valid:</th>
<th>MenuAttribute:</th>
<th>Valid:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MenuFunction</td>
<td>✓</td>
<td>MenuMarkState</td>
<td></td>
</tr>
<tr>
<td>MenuId</td>
<td></td>
<td>MenuModeFunction</td>
<td></td>
</tr>
<tr>
<td>MenuIndex</td>
<td>✓</td>
<td>MenuSelectState</td>
<td></td>
</tr>
<tr>
<td>MenuInit</td>
<td></td>
<td>MenuShortKey</td>
<td>✓</td>
</tr>
</tbody>
</table>

Example This sub menu has the same definition as the menu at page 103 except that it has a SubMenu data constructor rather than the Menu data constructor and a different title. The left picture shows the sub menu when not selected by the user, the right picture when selected by the user:

submenu
  = SubMenu "Sub Menu"
    (   MenuItem "Open..." [MenuShortKey 'o']
        :+: MenuItem "Close" [MenuSelectState Unable
                                ,MenuShortKey 'w'
                             ]
    )
    :+: MenuSeparator []
    :+: MenuItem "Quit" [MenuShortKey 'q']
) []

8.2 Menu glue

In the previous section the standard set of menus and menu elements has been discussed. This list is not complete. In the library module `StdMenuElementClass` (Appendix A.21) a number of additional instances are defined, namely the type constructors `:+:`, `ListLS`, `NilLS`, and `AddLS`, `NewLS` (their definition can be found in Appendix A.14). These additional instances are required to glue menu elements. They are treated below.

8.2.1 `:+:`

The most common constructor to glue menu elements is `:+:`. Its type constructor definition and `MenuElements` class instance declaration are as follows:

```haskell
:+: :: t1 t2 ls cs
   = (:+:) infixr 9 (t1 ls cs) (t2 ls cs)

instance MenuElements ((:+:) m1 m2) | MenuElements m1 & MenuElements m2
instance PopUpMenuElements ((:+:) m1 m2) | PopUpMenuElements m1 & PopUpMenuElements m2
```

Assume that we have two `MenuElements` (or `PopUpMenuElements`) instances `m1` and `m2`, working on the same local state of type `ls` and context state `cs`. Now the glued expression `m1:+:m2` is also a `MenuElements` (or `PopUpMenuElements`) instance working on the same local state and context state. Because `:+:` is right associative, the expression `m1 :+: m2 :+: m3` should be read as `m1 :+: (m2 :+: m3)`.

As an example, consider the following two menu elements:

```haskell
item  = MenuItem "Hello there!" []
separator = MenuSeparator []
```

Now the expressions `item :+: separator` and `(separator :+: item)` are legal combinations.

8.2.2 ListLS and NilLS

The `:+:` type constructor is not always the most appropriate glue. When constructing sets of menu elements of the same type, it is much more convenient to use lists and list comprehensions. When constructing an unknown number of menu elements, it is even impossible to use `:+:`. Again, lists provide more flexibility. List-like glue is provided by the type constructor `ListLS`. The type constructor `NilLS` is a shorthand for `ListLS []`. It can also be conveniently used to state that a sub menu or top level menu has no menu elements. Here are the definitions.
Given a list of MenuElements (PopUpMenuElements) instances \( ms = [m_1 \ldots m_n] \), working on the same local state of type \( ls \) and context state \( cs \), then the expression \( \text{ListLS } ms \) is also a MenuElements (PopUpMenuElements) instance working on the same local state and context state.

### 8.2.3 AddLS and NewLS

The previously discussed glueing type constructors always glue menu elements that work on the same local state and context state. Two other glueing constructors are defined to extend and change the local state, AddLS and NewLS.

Given a MenuElements (PopUpMenuElements) instance \( m_1 \) that works on a local state of type \( ls \) and a context state of type \( cs \), one can add another MenuElements (PopUpMenuElements) instance \( m_2 \) that works on an extended local state of type \( \text{new}, ls \) and the same context state of type \( cs \). Let \( x \) be a value of type \( \text{new} \), then this is done by the expression \( m_1 :\#\{\text{addLS}=:x, \text{addDef}=:t * (\text{new}, ls) \ cs \} \).

Given a MenuElements (PopUpMenuElements) instance \( m_1 \) that works on a local state of type \( ls \) and a context state of type \( cs \), one can add another MenuElements (PopUpMenuElements) instance \( m_2 \) that works on a new local state of type \( \text{new} \) and the same context state of type \( cs \). Let \( x \) be a value of type \( \text{new} \), then this is done by the expression \( m_1 :\#\{\text{newLS}=:x, \text{newDef}=:t \text{ new cs} \} \).

In both cases the extended part of the local state and the new local state are encapsulated completely from the external context using existential quantification.

Section 8.5 contains an example of the use of NewLS.

### 8.3 The Windows menu

For programs using interactive processes that have the multiple document interface (for MDI processes, see Section 11.1) one special menu is added by the object I/O system to the menu system of such an interactive process. This is the Windows menu. This menu contains a number of commands to arrange the current set of visible windows, and a list of all window titles. The set of arrange commands is platform dependent, but contains at least the following three commands:
Cascade:
This command arranges all windows to have equal size. They will be placed in diagonal order: from the left top to the right bottom of the process window. The windows will always be shown completely inside the process window. If there are too many windows, the arrangements starts over again for the remaining windows, which will therefore overlap the other windows.

Tile Horizontally:
This command arranges all windows in rows without overlapping. It is tried to give all windows the same size.

Tile Vertically:
This command arranges all windows in columns without overlapping. It is tried to give all windows the same size.

Separated from these commands by a MenuSeparator is the list of open windows in lexicographical order on the window title. The currently active window is checked. Selecting any of these windows activates that window. (This causes the previously active window to become deactivated, and the new window to become active.)

8.4 Menu conventions

The use of menus provides application users with a consistent and uniform access to the available set of commands. In this section we discuss a number of conventions that are usually followed to increase the level of consistency.

8.4.1 Subsetting the available commands

In general when using an interactive application, the application will move through several states. In each state a particular subset of the complete set of available commands will be applicable to the user while the remaining commands should not be selected. A well designed application should make this clear to the user by subsetting the available commands.

The easiest way to subset commands is by disabling and enabling the menu elements that should be unselectable and selectable respectively. For this purpose the functions enableMenuElements and disableMenuElements (module StdMenuElement, Appendix A.20) are available to enable and disable individual menu elements. Complete menus can be enabled and disabled using the functions enableMenus and disableMenus (module StdMenu, Appendix A.17). This module also contains two functions to enable and disable the whole current set of menus: enableMenuSystem and disableMenuSystem.

Another way to subset commands is to have at all times only those menus and menu elements open that should be available. A disadvantage of this technique is that it can become complicated quickly, and that it leaves the user disoriented when the set of commands changes frequently.

8.4.2 Command conventions

In this section we discuss some frequently used conventions in applications with respect to commands.
Clipboard commands

Applications that support the use of the clipboard (Chapter 12) to *cut*, *copy*, and *paste* private and external data usually are found in an "Edit" menu. Conventions are:

**Cut:**
This command should be enabled only if the application is in a state that an object has been selected that can be transferred to the clipboard. Issuing this command should remove that object from its context and place it in the clipboard. Its name should be "Cut" and it should have the shortkey attribute 'x'.

**Copy:**
This command should be enabled only if the application is in a state that an object has been selected that can be transferred to the clipboard. Issuing this command should place it in the clipboard, but not remove it from its context. Its name should be "Copy" and it should have the shortkey attribute 'c'.

**Paste:**
This command should be enabled only if the clipboard contains an object that can be currently incorporated in the application. Issuing this command should read the clipboard and put that object in the application. Its name should be "Paste" and it should have the shortkey attribute 'v'.

Undo command

Applications that allow users to manipulate documents by sequences of commands can support an *undo* command. The *undo* command can also be undone by a *redo* command. These commands are usually found in an "Edit" menu. Conventions are:

**Undo:**
This command should be enabled only if the user has issued a sequence of commands that can be undone. The number of undoable commands depends on the sophistication of the application. The name of this command is "Undo" and it has the shortkey attribute 'z'.

**Redo:**
This command should be enabled only if a (sequence of) *undo* command has been issued. At each selection it restores the changes of the *undo* command. The name of this command is "Redo" and it has the shortkey attribute 'y'.

Document commands

The document commands are frequently found commands to create new documents, open existing documents, and save and close open documents. These commands are usually found in a "File" menu. Conventions are:

**New:**
This command should be enabled only if the application can modify a new document. Issuing this command should create or reuse a window containing the new document. Its name should be "New" and it should have the shortkey attribute 'n'.
Open:
This command should be enabled only if the application can modify an additional, existing document. Issuing this command should give the user the opportunity to search for a file that will be opened by the application. For this purpose the `StdFileSelect` function `selectInputFile` can be used (Appendix A.10). The name of the command should be "Open..." and it should have the shortkey attribute 'o'.

Close:
This command should be enabled only if the application has an open document window or dialogue. Issuing this command should close the currently active window or dialogue. It is good programming practice to check if the document has been recently saved. If this is not the case, then the user should be asked if the document should be saved before closing. The name of this command should be "Close" and it should have the shortkey attribute 'w'.

Save:
This command should be enabled only if the currently active document window version differs from a (possibly not present) file version. Issuing this command should save the current state of the document in the active window to file. If there is no file associated yet, then the application should first ask for a file name. For this purpose the `StdFileSelect` function `selectOutputFile` can be used (Appendix A.10). The name of the command should be "Save" and it should have the shortkey attribute 's'.

Save As:
This command should be enabled only if the application has an open document window. Issuing this command should give the user the possibility to browse the file system and provide a file name. For this purpose the `StdFileSelect` function `selectOutputFile` can be used (Appendix A.10). The name of the command should be "Save As..."

Quit or Exit command
Users can leave an application using the `quit` or `exit` command. A user should always be allowed to leave the application. It is good programming practice to check if there are any unsaved documents in the application. If this is the case then the user should be asked if these documents should be saved before closing. The name of the quit command is usually "Quit" (with shortkey attribute value 'q') or "Exit". This command is usually found in a "File" menu.

8.5 Example: a small menu system

In this section we show a program that creates an interactive process with a multiple document interface. It can serve as a framework for writing your own multiple document interface programs. The interactive process contains one menu, `File`, which contains three commands: a command to open a new window, `New`; a command to close the active window, `Close`; and a command to leave the application, titled `Quit`. The `New` and `Quit` commands are always available. The `Close` command will be subsetted using enabling and disabling (as described in Section 8.4.1). We start with the commands, and proceed with the menu and process definition.

The `New` command is a `MenuItem` with a local state of type `Int`. The local state is used to give every new window a new title by appending the current value to the
string "Window". The initial value is one. Here is the definition of the command:

```java
{ newLS = 1,
  newDef = MenuItem "&New" [MenuShortKey 'n', MenuFunction new]
}
```

The callback function of the New command, new, uses the local integer state. For each successfully created window the value is incremented. In addition, the Close command is enabled. For now, assume that the Close command is identified by the Id value closeid. If the window can not be created a notice is opened, using the notice library developed in Section 6.8.1.

```java
new :: (Int, PST .1) -> (Int, PST .1)
new (i, pst)
  # (error, pst) = openWindow Void window pst
  | error<>NoError
    # notice = Notice ["MDI could not open new window"]
            (NoticeButton "Ok" id)
            []
    = (i, openNotice notice pst)
  | otherwise
    = (i+1, appPIO (enableMenuElements [closeid]) pst)
```

The Close command is identified by the Id closeid. The application starts with no windows, so its initial SelectState is Unable:

```java
MenuItem "&Close" [MenuShortKey 'w', MenuFunction (noLS close), MenuId closeid, MenuSelectState Unable]
```

The callback function close closes the active window with the StdWindow function closeActiveWindow. It is then checked if there still are open windows. If this is not the case, then close should disable the Close command. The function getWindowsStack returns the list of Ids of all currently open windows. So if this list is empty, then there are no more windows open. We get the following definition:

```java
close :: (PST .1) -> PST .1
close pst
  # pst = closeActiveWindow pst
  # (rest, pst) = accPIO getWindowsStack pst
  | isEmpty rest = appPIO (disableMenuElements [closeid]) pst
  | otherwise = pst
```

Because we are not interested in the window details, we keep its implementation very simple. It contains no controls, which expressed by the value NILLS. If the user requests to close the window, the Close command callback function close is evaluated. The look of the window draws the current view frame rectangle and the lines between the diagonally opposite corner points.

```java
window = Window ("Window "+toString i)
  NILLS
  [ WindowClose (noLS close)
  , WindowViewSize {w=300,h=300}
```
The Quit command has a straightforward definition:

```
MenuItem "&Quit" [MenuShortKey 'q',MenuFunction (noLS quit)]
```

Its callback function quit asks the user to confirm this choice. For this purpose we use another notice. If the user confirms the choice, then the application is terminated, otherwise nothing happens. Figure 8.1 shows the notice.

```
quit :: (PSt .1) -> PSt .1
quit pst
 = openNotice notice pst
where

  notice = Notice ["Do you really want to quit?"]
  (NoticeButton "&Ok" (noLS closeProcess))
  [NoticeButton "&Cancel" id]
```

The definition of the File menu simply glues the definitions of the commands as described above into one menu definition.

```
menu = Menu "&File"
 ( { newLS =1
   , newDef=MenuItem "&New" [MenuShortKey 'n',MenuFunction new]
   }
 :+: MenuItem "&Close" [MenuShortKey 'w',MenuFunction (noLS close)
   ,MenuId closeid, MenuSelectState Unable
   ]
 :+: MenuSeparator []
 :+: MenuItem "&Quit" [MenuShortKey 'q',MenuFunction (noLS quit)]
   []
```

The remaining details that need to be arranged are the creation of the Id value closeid, the creation of the menu, and the creation of the parent interactive process of the menu. We conveniently reuse the quit function to handle user requests to close the interactive process by setting the (ProcessClose quit) attribute. Here is the full program code.
module MDI

// ****************************************************************************
// Clean tutorial example program.
//
// This program creates a Multiple Document Interface process with a Window menu.
// ****************************************************************************

import StdEnv, StdIO
import notice

Start :: "World" -> "World"
Start world
  # (id,world) = openId world
  - startIO MDI Void (initialise id) [ProcessClose quit] world

quit :: (PSt .1) -> PSt .1
quit pst
  = openNotice notice pst
where
  notice = Notice ["Do you really want to quit?"]
    (NoticeButton "&Ok" (noLS closeProcess))
    (NoticeButton "&Cancel" id)

initialise :: Id (PSt .1) -> PSt .1
initialise closeid pst
  # (error,pst) = openMenu 0 menu pst
  | error<NoError abort "MDI could not open File Menu"
  | otherwise = pst
where
  menu= Menu "&File"
    ( { newLS =-1
        , newDef=MenuItem "&New" [MenuShortKey 'n',MenuFunction new]
    }
      :+ MenuItem "&Close" [MenuShortKey 'w',MenuFunction (noLS close)
        ,MenuId closeid
        , MenuSelectState Unable]
      :+ MenuSeparator []
      :+ MenuItem "&Quit" [MenuShortKey 'q',MenuFunction (noLS quit)]

new :: (Int,PSt .1) -> (Int,PSt .1)
new (i,pst)
  # (error,pst) = openWindow Void window pst
  | error<NoError
    # notice = Notice ["MDI could not open new window"]
      (NoticeButton "Ok" id)
      []
    = (i,openNotice notice pst)
  | otherwise
    = (i+1,appPIO (enableMenuElements [closeid]) pst)
where
  window = Window ("Window "++toString i)
    NILS
    [ WindowClose (noLS close)
      , WindowViewSize {x=300,h=300}
      , WindowLook True look
    ]

look :: SelectState UpdateState +Picture -> +Picture
look _ {newFrame=frame1:{corner1,corner2}} picture
  # picture = unfill frame1 picture
  # picture = draw frame1 picture
  # picture = drawLine corner1 corner2 picture
  = drawLine {corner1 & x=corner2.x} {corner2 & x=corner1.x} picture

close :: (PSt .1) -> PSt .1
close pst
  # pst  = closeActiveWindow pst
  # (rest, pst)  = accPST getWindowsStack pst
  | isEmpty rest  = appPST (disableMenuElements [closeId]) pst
  | otherwise  = pst
Chapter 9

Timers

Timers provide interactive programs with a tool to let actions occur at regular time intervals. These actions respond to timer events, and so they can be properly defined as callback functions. Typical examples of timer applications are blinking cursors, clocks, and time-out mechanisms.

The definition types of timers can be found in module StdTimerDef, Appendix A.44. The main type definitions are as follows:

```plaintext
:: Timer t ls pst
  = Timer TimerInterval (t ls pst) [TimerAttribute *(ls,pst)]
:: TimerInterval
  ::= Int

:: TimerAttribute  st
  = TimerFunction (TimerFunction st)
  | TimerId  Id
  | TimerInit  (IdFun st)
  | TimerSelectState SelectState
:: TimerFunction  st
  ::= NrOfIntervals -> st -> st
```

A TimerInterval is an integer value that must be at least zero. The time unit is platform dependent and is defined by the function ticksPerSecond (module StdSystem, Appendix A.37).

The TimerAttributes are the following:

**TimerFunction:**
This attribute is the callback function that is evaluated when a timer event is handled. Its first argument is the number of whole timer intervals that have elapsed since its previous evaluation, so this value is at least 1. If the timer interval is zero, then this number is always 1. Enabling a timer from a disabled state resets the last evaluation time.

**TimerId:**
This attribute identifies the timer. If you do not provide a TimerId then timer can not be modified or closed by the program (except by closing the parent interactive process).

**TimerInit:**
This attribute defines an action that should be performed immediately after
opening the timer. This is equivalent to the window and menu initialisation actions (sections 6.2.1 and 8.1.1). If no TimerInit attribute is provided, no additional action is performed.

**TimerSelectState:**
This attribute defines whether the timer will respond to timer events (Able) or not (Unable). The default value is Able.

The Timer type constructor is parameterised with a type constructor variable. Analogous to menus that contain menu elements, timers can contain timer elements. The instances must be member of the TimerElements class. This is expressed by the Timers timer creation member function, openTimer which can be found in module StdTimer (Appendix A.42):

```markdown
class Timers tdef where
  openTimer :: .ls !(tdef .ls (PSt .1)) !(PSt .1)
          -> (!ErrorReport, !PSt .1)

instance Timers (Timer t) | TimerElements t
```

Currently, the instances of the TimerElements class are receivers and the usual glueing type constructors that we have already encountered for controls (Section 7.2) and menu elements (Section 8.2), namely :+:, ListLS, NilLS, and AddLS, NewLS. The receiver instances are declared in module StdTimerReceiver (Appendix A.46). Receivers are handled in Chapter 10. The glueing constructors are declared in the same module StdTimerElementClass that contains the TimerElements class (Appendix A.45).

### 9.1 Examples

In this section we give some examples to illustrate the use of timers.

#### 9.1.1 Expanding circles

In this example we create a program that uses a timer to continuously draw a number of expanding concentric circles in a window. It also opens a menu containing only the quit command. Figure 9.1 shows the program in action. We discuss these object I/O components in reverse order (menu, window, timer).

The menu definition is very simple, as it contains only one command to terminate the example program. Recall that to terminate an interactive process the StdProcess function closeProcess must be used. The StdIOBasic function noLS suitably turns closeProcess into the desired type. Here is the menu definition.

```markdown
mdef = Menu "&Circles"
  (MenuItem "&Quit" [MenuFunction (noLS closeProcess)
                   ,MenuShortKey 'q'
                   ]
  ) []
```

The window is also simple. The initial view frame size of the window in which the circles are to be drawn is windowEdge by windowEdge (200 in the example). To
ease drawing, we take care that the view domain of the window has the origin zero exactly in the center of the view domain. This can be done conveniently by defining the WindowViewDomain attribute to be:

\[
\text{viewDomain} = \{ \text{corner1} = \{x= \text{windowEdge}/2, y= \text{windowEdge}/2\} \\
\quad, \text{corner2} = \{x= \text{windowEdge}/2, y= \text{windowEdge}/2\} \}
\]

The window does not contain controls (expressed by the Controls type instance NilLS). It is identified by the Id value wid. The window definition is as follows:

\[
\text{wdef} = \text{Window "Circles" NilLS} \\
\quad[\text{WindowId wid} \\
\quad, \text{WindowViewSize (rectangleSize viewDomain)} \\
\quad, \text{WindowViewDomain viewDomain}]
\]

The timer draws a number of concentric circles that have an increasing radius. For this purpose it uses a local state of the following type and initial value:

\[
\text{:: TimerState} = \{ \text{nrCircles} :: \text{Int} \\
\quad, \text{equiDistance} :: \text{Int} \\
\quad, \text{minRadius} :: \text{Int} \}
\]

\[
\text{initTimerState} = \{ \text{nrCircles}=4, \text{equiDistance}=2, \text{minRadius}=0 \}
\]

The \text{nrCircles} field contains the number of circles that are drawn. The \text{equiDistance} field is the difference of radius between two neighbouring circles. The \text{minRadius} field keeps track of the radius of the smallest visible circle.

The timer interval is set to a twentieth of a second (ticksPerSecond/20). The timer contains no timer elements. Its definition is as follows:

\[
\text{tdef} = \text{Timer (ticksPerSecond/20) NilLS [TimerFunction timer]}
\]
The timer function \text{timer} will be evaluated by the object I/O system every twentieth of a second (if possible). The timer function actually ignores the number of elapsed intervals and simply draws the next sequence of circles. There are two cases to distinguish:

- If the smallest circle still fits entirely inside the window (tested by $\text{minRadius} < \text{windowEdge}/2$), then \text{timer} undraws the smallest circle and draws the new circle which should have a radius equal to $\text{minRadius} + \text{nrCircles} \times \text{equiDistance}$. Finally, the \text{minRadius} field is changed to reflect the fact that the smallest visible circle now has radius $\text{minRadius} + \text{equiDistance}$.

- If the smallest circle does not fit entirely inside the window, then \text{timer} completely unfills the window. By setting the new local \text{TimerState} back to \text{initTimerState} the circles are drawn again from the center.

\begin{verbatim}
\text{timer} \_ (\text{lst}=:\{\text{nrCircles, equiDistance, minRadius}\}, \text{pst}) |
\text{minRadius}\text{<}\text{windowEdge}/2
# \text{lst} = \{\text{lst & minRadius=\text{minRadius}+\text{equiDistance}}\}
\text{newRadius} = \text{minRadius} + \text{nrCircles} \times \text{equiDistance}
# \text{pst} = \text{appPIO (appWindowPicture wid}
  (\text{draw} \{\text{oval\_rx=\text{newRadius}, oval\_ry=\text{newRadius}}\}
  \text{undraw} \{\text{oval\_rx=\text{minRadius}, oval\_ry=\text{minRadius}}\}
) \text{pst}
= (\text{lst, pst})
| \text{otherwise}
# \text{pst} = \text{appPIO (appWindowPicture wid (unfill \text{viewDomain})) pst}
= (\text{initTimerState, pst})
\end{verbatim}

The last details that remain to be defined are the actual opening of the menu, window, and timer, the opening of the interactive process, and the creation of \text{wid}. For completeness we show the complete program code.

\begin{verbatim}
module circles

// **************************************************
// Clean tutorial example program.
// This program creates a window that displays growing concentric circles.
// For this purpose it uses a timer.
// **************************************************

import StdEnv, StdIO

:: TimerState
  = \{ \text{nrCircles} :: \text{Int}
      , \text{equiDistance} :: \text{Int}
      , \text{minRadius} :: \text{Int}
  \}

\text{Start} :: *\text{World} \rightarrow *\text{World}
\text{Start world}
  = \text{circles (openId world)}

\text{circles} :: \{\text{Id, World}\} \rightarrow *\text{World}
\text{circles (wid, world)}
  = \text{startIO \text{SDI}
    Void}
    \text{snd o seqList [openWindow Void \text{wdef}}
    , \text{openMenu Void \text{wdef}}
\end{verbatim}
In this example we create a program that uses three timers to track the elapsed time since program startup. The timers track the elapsed seconds, minutes, and hours respectively. A dialogue is used to provide visual feedback. Figure 9.2 shows the application in action. We first have a look at the dialogue, and then the three timers.

![Clock](image)

Figure 9.2: The timing program in action.

The dialogue `ddef` simply uses `TextControls` to display the hours, minutes, and seconds. These controls are identified by the `Id` values `hoursId`, `minutesId`, and...
secondsId respectively. LayoutControls are used to get the layout in the desired way. Observe the use of list comprehensions and the ListLS Controls class instance. In this example closing the dialogue also terminates the application. Here is the dialogue definition:

ddef
def "Clock"
  ( LayoutControl
    ( ListLS [TextControl text [ControlPos (Left,zero)]
      \ text<-["Hours:","Minutes:","Seconds:"]
    ]
  ) []
:+: LayoutControl
  ( ListLS [TextControl "0" [ControlPos (Left,zero)
    ,ControlId id
    ,ControlWidth (ContentWidth "00")
  ]
    \ id<-[hoursId,minutesId,secondsId]
  )
) [WindowClose (noLS closeProcess)]

Because the operation of each of the three timers is very similar, we use one function tdef to define them and parameterise it with their respective TimerIntervals, dt.

The timers do not contain any timer elements. Each timer has a local integer state (initially zero) that keeps the current number of evaluated time units modulo their maximum number, maxunit. The timer function tick uses the NrOfIntervals parameter. Given the current number of evaluated time units in its local integer state, each timer function adds the NrOfIntervals value nrElapsed to it. Depending on dt, the proper modulo value is taken (using maxunit). This new value is the new local state and is also drawn in the dialogue by setting the text of the corresponding text control.

tdef dt
def Timer dt NilLS [TimerFunction tick]
where
  tick nrElapsed (time,pst)
    # time = (time+nrElapsed) mod maxunit
    = (time,appPIO (setControlText id (toString time)) pst)
    (id,maxunit) = if (dt==second) (secondsId,60)
                  (if (dt==minute) (minutesId,60)
                   (hoursId, 24))

The last details that remain to be defined are the actual opening of the three timers, the dialogue, the opening of the interactive process, and the creation of the proper Ids. For completeness we show the complete program code.

module clock

// ******************************************************************************
// Clean tutorial example program.
//
// This program creates a window that tracks the elapsed time since startup.
// For this purpose it uses three timers to track the seconds, minutes, and hours
// separately.
import StdEnv, StdIO

:: DialogIds
  = { secondsId :: Id
      , minutesId :: Id
      , hoursId :: Id
    }

second  ::= ticksPerSecond
minute  ::= 60*second
hour    ::= 60*minute

openDialogIds :: env -> (DialogIds, env) | Ids env
openDialogIds env
  # (secondsId, minutesId, hoursId :: env) = openIds 3 env
  = (secondsId=secondsId, minutesId=minutesId, hoursId=hoursId), env)

Start :: *World -> *World
Start world
  # (dialogIds, world) = openDialogIds world
  = startIO NDI Void (initialise dialogIds) [] world

initialise :: DialogIds (PSt .1) -> (PSt .1)
initialise {secondsId, minutesId, hoursId} pst
  # (errors, pst) = seqList [ openTimer 0 (tdef timerInfo)
      \timerInfo<[second, minute, hour]
    ] pst
    | any (<<>) NoError errors
    - closeProcess pst
    # (error, pst) = openDialog Void ddef pst
    | error<>NoError
    - closeProcess pst
    | otherwise
    - pst

where
tdef dt
  = Timer dt NilLS [TimerFunction tick]

where
tick nElapsted (time, pst)
  # time = (time+nElapsted) mod maxunit
  = (time, appPIO (setControlText id (toString time)) pst)

(id, maxunit) = if (dt==second) (secondsId, 60)
  (if (dt==minute) (minutesId, 60)
   (hoursId, 24))

ddef= Dialog "Clock"
  | LayoutControl
  | { ListLS [ TextControl text [ControlPos [Left, zero]]
      \text<"["Hours":","Minutes":","Seconds:"]
    ]
  }
  | []
  +: LayoutControl
  | { ListLS [ TextControl "0" [ControlPos [Left, zero]]
      ,ControlId id
      ,ControlWidth (ContentWidth "00")
      \id<[hoursId, minutesId, secondsId]
    ]
  }
  | []
  | [WindowClose (noLS closeProcess)]
Chapter 10

Receivers

All the interactive object I/O components discussed so far have in common that the events to which they respond are abstract. In this context abstract means that it is not specified in detail what concrete events cause a specific callback function to be evaluated. For instance, a callback function associated with a MenuItem (Section 8.1.4) is evaluated when it has been selected by the user. How this selection takes place is not specified.

In this section we discuss an interactive object I/O component that responds to program defined events, or rather messages. This component is the receiver. It plays an important role in the construction of interactive components. There are no restrictions on the kind of messages that can be sent or received, provided that they are type correct. The latter is obtained by using special identification values for receivers, the receiver ids (Chapter 4).

There are also receivers that are able to receive data from other programs via a network. These receivers are not topic of this chapter but of Chapter 14.

We will first have a look at the definition of receivers in Section 10.1. Receivers can be opened as top level object I/O components, but also as elements of windows, dialogues, menus, and timers. This is discussed in Section 10.2. Knowing how to define and open receivers, we show which functions are available to send messages in Section 10.3. Section 10.4 contains a number of examples to demonstrate the use of receivers.

10.1 Receiver definitions

There are two kinds of receivers. Uni-directional receivers respond only to messages. Bi-directional receivers respond to messages and also reply with a message.

The types needed to define receivers can be found in the module StdReceiverDef, Appendix A.34.

A uni-directional receiver that responds to messages of type msg and that has a local state of type ls and process state of type pst) is defined by an expression of type (Receiver msg ls pst):

:: Receiver msg ls pst
   = Receiver (RId msg) (ReceiverFunction msg *(1st,pst))
       [ReceiverAttribute *(1st,pst)]
:: ReceiverFunction msg st

125
A bidirectional receiver that responds to messages of type `msg` and returns a response message of type `resp` and that has a local state of type `ls` and process state `pst`) is defined by an expression of type `(Receiver2 msg resp ls pst):

```plaintext
:: Receiver2 msg resp ls pst
   = Receiver2 (R2Id msg resp) (Receiver2Function msg resp *(ls,pst))
     [ReceiverAttribute *(ls,pst)]
```

:: Receiver2Function msg resp st
::= msg -> st -> (resp, st)

The set of receiver attributes is currently limited to the `ReceiverSelectState` which default value is `Able`.

The receiver attributes are defined as follows:

```plaintext
:: ReceiverAttribute st
   = ReceiverInit (IdFun st)
     | ReceiverSelectState SelectState
     | ReceiverConnectedReceivers [Id]
```

The `ReceiverInit` attribute defines an action that should be performed immediately after opening the receiver. This is equivalent to the window, menu, and timer initialisation actions (sections 6.2.1, 8.1.1, and 9). If no `ReceiverInit` attribute is provided, no additional action is performed. The `ReceiverSelectState` attribute indicates whether the receiver responds (`Able`) or ignores (`Unable`) messages. Its default value is `Able`. The `ReceiverConnectedReceivers` attribute is discussed in Section 14.

### 10.2 Receiver creation

Receivers can be opened as top level interface elements. This is done in the usual, overloaded way. Module `StdReceiver`, Appendix A.32) declares the type constructor class `Receivers` with the top level receiver creation member function `openReceiver`. Uni-directional and bi-directional receivers are proper instances of this class.

```plaintext
class Receivers rdef where
   openReceiver :: .ls !*(rdef .ls (PSt .1)) !(PSt .1)
                  -> (!ErrorReport,!PSt .1)

   ...
```

instance Receivers (Receiver msg)
instance Receivers (Receiver2 msg resp)

Receivers can also be opened as elements of windows, dialogues, menus, and timers. So, for instance in a window one can not only add the set of controls as discussed in Chapter 7 but also receivers. This is accomplished by declaring receivers to be instances of the `Controls` type constructor class in module `StdControlReceiver` (Appendix A.32). Receivers are declared to be instances of `menu elements` in module `StdMenuReceiver` (Appendix A.22), and `timer elements` in module `StdTimerReceiver` (Appendix A.46).
10.3 Message passing

In contrast with the previously discussed object I/O components, receivers must have an identification value. The message passing functions require this identification value to ensure that a message of the correct type is sent. The message passing functions can be found in module StdReceiver, Appendix A.32.

All message passing functions return a report about the message passing action. This report is an algebraic type SendReport, which has the following alternatives:

```plaintext
:: SendReport
  = SendOk
  | SendUnknownReceiver
  | SendUnableReceiver
  | SendDeadlock
  | OtherSendReport !String
```

For all functions, the alternative value `SendOk` is returned in case message passing was successful. The alternative `SendUnknownReceiver` is returned in case the indicated receiver is not open at the moment of sending the message. The other `SendReport` alternatives are discussed below.

We start with message passing to uni-directional receivers in Section 10.3.1. Bi-directional message passing is discussed in Section 10.3.2.

10.3.1 Uni-directional message passing

There are two functions a programmer can use to send a message to a uni-directional receiver: `asyncSend` and `syncSend` which have the same function types.

```plaintext
asyncSend :: !(RId msg) msg !PSt .1) -> ![SendReport, !PSt .1]
syncSend :: !(RId msg) msg !PSt .1) -> ![SendReport, !PSt .1]
```

Using `asyncSend`, a message is placed at the end of the asynchronous message queue of the indicated receiver and will, at some point, be handled by that receiver. It is unspecified when that event occurs. Asynchronous messages that are sent in sequence will be evaluated in that sequence. Asynchronous message passing only fails in case the indicated receiver is not open, as discussed above. You should observe that successfully queueing an asynchronous message does not guarantee that the message will be handled. The receiver might be disabled or closed before all of its messages have been handled.

If you want to enforce a receiver to handle a message, you should use `syncSend`. This function does not place the message in the message queue of the indicated receiver, but instead applies the receiver function of that receiver to the message. This implies that `syncSend` has to switch context because the indicated receiver is part of another object I/O component. Another consequence is that synchronous messages may overtake asynchronous messages. Synchronous messages that are sent in sequence will be evaluated in that order.

Sending a synchronous message fails in case the indicated receiver does not exist. If the indicated receiver does exist, but its ReceiverSelectState attribute is `Unable`, then the message is also not handled. In this case, the `SendReport` alternative `SendUnableReceiver` is returned. Finally, even if the indicated receiver exists and is `Able` communication can still fail. This is possible when the indicated receiver itself
is involved in a synchronous message passing transaction and waiting for termination. If this transaction involves the original receiver, then a deadlock situation is detected. In that case, syncSend also fails and returns with SendDeadlock.

Examples of uni-directional message passing are given in Section 10.4. The first example, in Section 10.4.1, demonstrates the use of asynchronous message passing, while the second example, in Section 10.4.2, demonstrates synchronous message passing.

### 10.3.2 Bi-directional message passing

Bi-directional message passing is synchronous. A message is sent using the function syncSend2:

```
syncSend2 :: !R2Id msg resp msg !(PSt .1)
  -> (!SendReport, !Maybe resp), !(PSt .1)
```

Analogous to syncSend, syncSend2 locates the indicated bi-directional receiver and applies the argument message immediately to the corresponding receiver function. If this receiver could be found and happens to be Able, evaluation of the receiver function will yield a response message resp. In this case the SendReport result of syncSend2 is SendOk, and the response value is returned as (Just resp). In all exceptional cases, there is no response value and Nothing is returned. To evaluate the receiver function, syncSend2 has to switch context as well.

Bi-directional receivers can be used to retrieve local encapsulated data. Example 10.4.3 demonstrates this.

### 10.4 Examples

In this section we give some examples to illustrate the use of receivers.

#### 10.4.1 Talk windows

In this example we create a program that uses receivers to send keyboard input from one window to another window, and vice versa. This results in a talk like application (although it is not very useful as a talk application because it runs on one computer). Figure 10.1 shows the application in action.

The initialisation action of the talk program, defined by the function initialise, first creates a menu that has only one quit menu item. This command ensures that we can always terminate the application. Then two RId values are generated that will be used to identify the two receivers. The function openTalkWindow is then applied twice to create the two windows. Its parameters are the name of the window and the two receiver ids. The first receiver id parameter identifies the private receiver, while the second identifies the other receiver.

```
initialise :: (PSt .1) -> PSt .1
initialise pst
   # menu       = Menu "&Talk"
               (MenuItem "&Quit" [MenuShortKey 'q'
                       ,MemuFunction (noLS closeProcess) ]
```

openTalkWindow creates a window in which the user can type text, using an edit control, and see the messages of the other talk window, also by means of an edit control. The messages of the other talk window are of course handled by a receiver. The first edit control is identified by an Id value inId. The user keyboard input is handled by function input. This function is parameterised with the Id of the edit control and the RId of the receiver of the other window. Because we only want to send new input to the other window receiver, the keyboard filter function inputFilter filters out key up events. The edit control shares the window view frame real estate with the other edit control. When the window is being resized, it adopts its width to the new width of the window and consumes half the height. This is expressed by the function resizeHalfHeight \_\_ \{w,h\} = \{w=w,h=h/2\}.

input = EditControl "" (PixelWidth (hmm 50.0)) 5
    [ ControlId inId
      , ControlKeyboard inputfilter Able (noLS1 (input inId you))
      , ControlResize resizeHalfHeight
    ]
inputfilter :: KeyboardState -> Bool
inputfilter keystate = getKeyboardStateKeyState keystate<>KeyUp

The keyboard input function input first gets the current state of its parent window, using the StdControl library function getParentWindow. From this value the current content of its edit control can be retrieved, using the function getControlText. This new content, which is a String, is being sent asynchronously to the other receiver.

input :: Id (RId String) KeyboardState (PSt .1) -> PSt .1
input inId you pst
    # (Just wst, pst) = accPIO (getParentWindow inId) pst
    # text = fromJust (snd (getControlText inId wst))
    = snd (asyncSend you text pst)

The second edit control is used to display the received text from the other window receiver. Because it is used only for output its SelectState is Unable. To be able to change its text content it is identified by the Id value outId. Its position is below the first edit control. Its resize behaviour is identical to that of the first edit control. Here is its definition:

```
output = EditControl "" (PixelWidth (hmm 50.0)) 5
    [ ControlId outId
    , ControlPos (BelowPrev, zero)
    , ControlSelectState Unable
    , ControlResize resizeHalfHeight ]
```

The final component of the window is the receiver that obtains the new string content of the other window input edit control. Its receiver function receive is parameterised with the Id of the output edit control. For every string message received from the other talk window, it replaces the current content of the output edit control with the received text. This is done using the function setControlText. The function setEditControlCursorPosition makes sure that the end of the text is visible.

```
receiver = Receiver me (noLS1 (receive outId)) []
receive :: Id String (PSt .1) -> PSt .1
receive outId text pst::{io}
    # io = setControlText outId text io
    # io = setEditControlCursorPosition outId (size text) io
    = {pst & io=io}
```

These are the three components of the window. The only task of openTalkWindow is to create the two Id values that identify the input and output edit controls, and open the window containing the two edit controls and the receiver control. We also want the initial window view frame size to be equal to the width and added height of the two edit controls. Because the height of the edit controls is given indirectly (by specifying that it should be as high as five text lines) we have to ask the system to calculate the size of the edit controls. This is done by the StdControl function controlSize. Its first argument is the Controls instance which size has to be calculated. The second argument, a boolean, must be true if the controls are supposed to be opened in a window and false if they are to be opened in a dialogue. This value is required because of the different platform margins and item spaces. If you want to provide your own horizontal and vertical margins and item spaces then this can be done in the following three arguments. Choosing Nothing results in the default values. The result size of controlSize can now be used correctly as the initial view frame size of the window. Here is the definition of openTalkWindow.

```
openTalkWindow:: String (RId String) (RId String) (PSt .1) -> PSt .1
openTalkWindow name me you pst
    # (inId, pst)= accPIO openId pst
```
For completeness the whole program is shown here.

```haskell
module talk

// Clean tutorial example program.
//
// This program creates two windows that communicate with each other using message passing. Text that has been typed in one window is being sent to the other, and vice versa.

import StdEnv, StdIO

Start :: *World -> *World
Start world
    = startIO MDI Void initialise [] world
where
    initialise :: ([St,.l]) -> [St,.l]
    initialise pst
        # menu
            = Menu "&Talk"
                ( MenuItem "&Quit" [ MenuShortKey 'q'
                        , MenuFunction (noLS1 closeProcess)
                    ]
                )
        # (error,pst) = openMenu undef menu pst
        | error>NoError = abort "talk could not open menu."
        | otherwise
            # (a,pst) = accPIO openId pst
            # (b,pst) = accPIO openId pst
            # pst = openTalkWindow "A" a b pst
            # pst = openTalkWindow "B" b a pst
        = pst

openTalkWindow :: String (RId String) (RId String) ([St,.l]) -> [St,.l]
openTalkWindow name me you pst
    # (inId, pst) = accPIO openId pst
    # (outId,pst) = accPIO openId pst
    # input = EditControl "" (PixelWidth (hmm 50.0)) 5
```
10.4.2 Resetting the counter

In this example we extend the example counter in Section 7.2.3 (page 89) with a means to reset the counter to zero. We proceed in a bottom-up style: the counter control is a layout control extended with a receiver that, when it receives a message, will reset the counter to zero. The counter control encapsulates its local counter state. Out of the local state scope of the counter control we add button control that, when selected, sends a message to the receiver component of the counter control. The whole is being placed in a dialogue. Figure 10.2 gives a snapshot of the program.

![Counter](image)

Figure 10.2: The counter control with reset button.
The main component is of course the counter control, defined by `counter`. It encapsulates an integer local state with initial value `initcount`. We have changed its definition from the original one shown on page 89 to give another example of using control layout. It is also different in that it contains a receiver.

```plaintext
counter = { newLS = initcount,
   , newDef = LayoutControl
   (   EditControl (toString initcount)
       (PixelWidth (hmm 50.0)) 1
       [ControlSelectState Unable
           ,ControlId displayid
       ]
   )
   :+: ButtonControl "-"
   [ControlFunction (count (-1))
     ,ControlWidth (PixelWidth (hmm 25.0))
     ,ControlPos (BelowPrev,zero)
   ]
   :+: ButtonControl "+
   [ControlFunction (count 1 )
     ,ControlWidth (PixelWidth (hmm 25.0))
   ]
   :+: Receiver resetid reset []
       [ControlPos (Center,zero)
           ,ControlHMargin 0 0
           ,ControlVMargin 0 0
           ,ControlItemSpace 0 0
       ]
 }
```

The only purpose of the receiver is to reset the current local counter value to `initcount` and show this by changing the text of the edit control. Note that the receiver function `reset` is not interested at all in the message.

```plaintext
reset :: m (Int,PSt .1) -> (Int,PSt .1)
reset _ (._,pst)
   = ( initcount
     , appPIO (setControlText displayid (toString initcount)) pst
   )
```

The reset button is a straightforward `ButtonControl`. It is centered below the counter control. Its `ControlFunction` argument only has to send a message synchronously to the receiver component of the counter control. Because this component does not care about the message, the button function can be as bold to send the `StdMisc` library function `undef`. This function, when evaluated, aborts the application. This demonstrates that message passing is truly lazy in the message argument.

```plaintext
resetbutton
   = ButtonControl "Reset"
      [ControlFunction (noLS (snd o syncSend resetid undef))
         ,ControlPos (Center,zero)
      ]
```
The final details of the program are to generate the proper identification values and to create the initial process and dialogue. For completeness, the program code is given here.

```haskell
module counterreset

//******************************************************************************
// Clean tutorial example program.
//******************************************************************************
// This program defines a Controls component that implements a manually settable
// counter. A receiver is used to add a reset option.
//******************************************************************************

import StdEnv, StdIO

Start :: *World -> *World
Start world
  = startIO NDI Void initialise [] world
where
  initialise pst
    # (displayid, pst)  = accPIO openId pst
    # (resetid, pst)  = accPIO openRId pst
    # (error, pst)  = openDialog Void (dialog displayid resetid) pst
    error<\NoError
    = abort "counter could not open Dialog."
    otherwise
    = pst
dialog displayid resetid
  = Dialog "Counter" (counter :+: resetbutton) [WindowClose (noLS closeProcess)]
where
counter
  = { newLS = initcount
      , newDef = LayoutControl
        ( EditControl (toString initcount) (PixelWidth (hmm 50.0)) 1
          [ControlSelectState Unable
            ,ControlId displayid
          ]
        ) :+: ButtonControl "-" [ControlFunction (count (-1))
          ,ControlWidth (PixelWidth (hmm 25.0))
          ,ControlPos (BelowPrev,zero)
        ]
        :+: ButtonControl "+" [ControlFunction (count 1)
          ,ControlWidth (PixelWidth (hmm 25.0))
        ]
        :+: Receiver resetid reset []
        ) [ControlPos (Center.zero)
          ,ControlHMargin 0 0
          ,ControlVMargin 0 0
          ,ControlItemSpace 0 0
        ]
      }
where
  initcount  = 0

count :: Int (Int,PSt .1) -> (Int,PSt .1)
count dx (count,pst)
  = (count+dx,appPIO (setControlText displayid (toString (count+dx))) pst)
reset :: = (Int,PSt .1) -> (Int,PSt .1)
reset _ (._,pst)
  = (initcount,appPIO (setControlText displayid (toString initcount))) pst
resetbutton
  = ButtonControl "Reset"
    [ControlFunction (noLS (snd o syncSend resetid undef))
      ,ControlPos (Center.zero)
    ]
```

CHAPTER 10. RECEIVERS
10.4.3 Reading the counter

In this example we extend the counter example once more and add a dialogue that reads the local counter value. To be able to do this a bi-directional receiver is added to the counter. Figure 10.3 gives a snapshot of the program.

The first change is to extend the counter component with a bi-directional receiver component defined by the expression \((\text{Receiver2 readid read []})\), where \(\text{readid}\) is a \(\text{R2Id}\) identification value. The receiver function \(\text{read}\) is very straightforward; it returns the current local counter value without changing anything (also in this case \(\text{read}\) is not interested in the input message type):

\[
\text{read} :: \text{m (Int,PSt .1)} \rightarrow (\text{Int,(Int,PSt .1)})
\]

\[
\text{read} _\_ (\text{count,pst}) = (\text{count,}(\text{count,pst}))
\]

From the type of this function we can derive that the type of the \(\text{R2Id}\) value of its associated receiver must be \((\text{R2Id m Int})\).

The initialisation action opens another dialogue defined by \(\text{display}\) that obtains and shows the counter value on request. This request can be done by a user by pressing the button control labeled "Read". Again, an \(\text{Unable}\) edit control is used to display the read value.

\[
\text{display displayid readid} = \text{Dialog "Read"}
\]

When the "Read" button has been selected, its callback function \(\text{read}\) is evaluated. It sends a message to the bi-directional receiver component of the counter control using \(\text{syncSend2}\). If this action fails it aborts the program. Although this situation will never occur, this check has been added for program hygiene. In a successful communication, value will be \((\text{Just count})\), and it is this value that is then displayed in the display edit control of the dialogue.
The initialisation action creates the necessary identification values and the two dialogues. Here is the complete program code.

```haskell
module counterread

-- Clean tutorial example program.
-- This program defines a Controls component that implements a manually settable counter.
-- A bi-directional receiver is added to give external access to the counter value.
--
import StdEnv, StdIO

Start :: *World -> *World
Start world = startIO NDI Void initialise [] world
where
  initialise pst
      # (displayid, pst) = accPIO openId pst
      # (resetid, pst) = accPIO openRId pst
      # (readid, pst) = accPIO openR2Id pst
      # (error, pst) = openDialog Void (dialog displayid resetid readid) pst
          | error<*>NoError
            = abort "counter could not open counter dialog."
      # (displayid, pst) = accPIO openId pst
      # (error, pst) = openDialog Void (dialog displayid readid) pst
          | error<*>NoError
            = abort "counter could not open display dialog."
          | otherwise = pst
  dialog displayid resetid readid
      = Dialog "Counter" (counter:::resetbutton) [WindowClose (noLS closeProcess)]
where
counter
  = { newLS = initcount
   , newDef = LayoutControl
      ( EditControl (toString initcount) (PixelWidth (hms 50.0)) 1
        [ControlSelectState Unable
          ,ControlId displayid
        ]
      )
      :+: ButtonControl "-" [ControlFunction (count (-1))
        ,ControlWidth (PixelWidth (hms 25.0))
        ,ControlPos (BelowPrev,zero)
      ]
      :+: ButtonControl "+" [ControlFunction (count 1 )
        ,ControlWidth (PixelWidth (hms 25.0))
      ]
      :+: Receiver resetid reset []
      :+: Receiver2 readid read []
    ) [ControlPos (Center,zero)
        ,ControlHMargin 0 0
        ,ControlVMargin 0 0
        ,ControlItemSpace 0 0
      ]
```

read pst
  # ((error,value),pst)= syncSend2 readid undef pst
  | error<*>SendOk
    = abort "could not read counter value"
  | otherwise
    = appPIO (setControlText displayid
                (toString (fromJust value)))
            pst
where

\[
\begin{align*}
\text{initcount} &= 0 \\
\text{count} :: (\text{Int}, \text{PSt} .1) \rightarrow (\text{Int}, \text{PSt} .1) \\
\text{count dx} (\text{count}, \text{pst}) &= (\text{count+dx}, \text{appPIO (setControlText displayid (toString (count+dx))) pst}) \\
\text{reset} :: m (\text{Int}, \text{PSt} .1) \rightarrow (\text{Int}, \text{PSt} .1) \\
\text{reset} _{m} (\text{..., pst}) &= (\text{initcount}, \text{appPIO (setControlText displayid (toString initcount)) pst}) \\
\text{read} :: m (\text{Int}, \text{PSt} .1) \rightarrow (\text{Int}, (\text{Int}, \text{PSt} .1)) \\
\text{read} _{m} (\text{count}, \text{pst}) &= (\text{count}, (\text{count}, \text{pst})) \\
\end{align*}
\]

\[
\begin{align*}
\text{resetbutton} &= \text{ButtonControl "Reset"} \\
&= [\text{ControlFunction (noLS (snd o syncSend resetid undef))} \\
&\quad \text{ControlPos (Center, zero)}] \\
\text{display displayid readid} &= \text{Dialog "Read"} \\
&= (\text{EditControl "" (PixelWidth (hmm 50.0)) 1} \\
&\quad [\text{ControlSelectState Unable} \\
&\quad \quad \text{ControlId displayid}] \\
&\quad :: \text{ButtonControl "Read"} [\text{ControlFunction (noLS read)} \\
&\quad \quad \text{ControlPos (Center, zero)}] \\
&\quad ) [\text{WindowClose (noLS closeProcess)}] \\
\end{align*}
\]

where

\[
\begin{align*}
\text{read pst} &= # ((\text{error}, \text{value}), \text{pst}) = \text{syncSend2 readid undef pst} \\
&\mid \text{error} < \text{SendOk} \\
&\quad = \text{abort "could not read counter value"} \\
&\mid \text{otherwise} \\
&\quad = \text{appPIO (setControlText displayid (toString (fromJust value))) pst}
\end{align*}
\]
Chapter 11

Interactive processes

The examples discussed so far used the StdProcess library function startIO. This function creates one interactive process that engages in some graphical user interface actions with a user and then terminates, using the StdProcess function closeProcess. In this chapter we show how an interactive process can spawn new interactive processes that will run interleaved. It is also possible to create a number of processes at once.

We start the discussion by looking at the ways to define interactive processes in Section 11.1. This is followed by the functions to open interactive processes. Finally we give some examples to illustrate their application.

11.1 Defining interactive processes

All interactive processes are defined as an instance of the following type definition (module StdProcessDef, see Appendix A.29):

```plaintext
:: Process
 = E .1: Process
     DocumentInterface
     1
     (ProcessInit (PSt 1))
     [ProcessAttribute (PSt 1)]

:: DocumentInterface
 = NDI | SDI | MDI
```

The DocumentInterface argument of an interactive process specifies its kind of document interface. The document interface concept is based on the idea that an interactive process provides the user with an interface to manipulate documents (even if they don’t look like documents at all). Depending on the number of documents a process offers (zero, one, or many) the library provides the process with a platform dependent infrastructure. Three kinds of document interfaces are specified:

No Document Interface (NDI)

An interactive process with this document interface does not present a document to a user. It has no menus and no windows. It is typically used for ‘background’ interactive processes or very simple programs that have only a
dialogue. The interactive process is allowed to open dialogues, timers, and receivers.

**Single Document Interface (SDI)**

An interactive process with this document interface presents exactly one document at a time to the user. Usually all interactive components are presented within one window frame. The interactive process is allowed to open dialogues, timers, and receivers. At any time, at most one window can be open. It is allowed to close a window and then open a new one.

**Multiple Document Interface (MDI)**

An interactive process with this document interface can present an arbitrary number of documents to the user.

If some callback function or initialisation function attempts to open an interactive object that violates its document interface, then the ErrorReport result of that particular object opening function is ErrorViolateDI. When defining an interactive process, you should choose the most ‘minimal’ document interface that fits that process.

The interactive process type constructor introduces an initial local process state of some type \( l \) and encapsulates it using existential quantification. When the interactive process is created, the object I/O system creates the PSt record with for its \( ls \) field the value of this argument. Because of the limited sizes of the example programs so far, the value we have used so far was Void. In addition to the local process state value, the object I/O system creates an initial, empty I/O state of type IOSt.

Given the initial process state record, the object I/O system lets the interactive process initialise itself by applying the third argument of the Process type constructor which has type \((\text{ProcessInit} (\text{PSt} l))\).

\[
\text{ProcessInit pst} ::= \text{IdFun pst} \\
\text{IdFun st} ::= \text{st} \rightarrow \text{st}
\]

When this function has been evaluated, and the interactive process has not been terminated by the closeProcess function, the interactive process is in its running phase. It should be noted that it is unspecified when the initialisation of the process takes place: it is guaranteed only that this function is the first action on the process state.

The final argument of the Process type constructor are its attributes. These are the following:

\[
\text{ProcessAttribute st} = \text{ProcessActivate} (\text{IdFun st}) \\
| \text{ProcessDeactivate} (\text{IdFun st}) \\
| \text{ProcessClose} (\text{IdFun st}) \\
| \text{ProcessOpenFiles} (\text{ProcessOpenFilesFunction st}) \\
| \text{ProcessWindowPos} \text{ ItemPos} \\
| \text{ProcessWindowSize} \text{ Size} \\
| \text{ProcessWindowResize} (\text{ProcessWindowResizeFunction st}) \\
| \text{ProcessToolbar} [\text{ToolbarItem st}] \\
| \text{ProcessNoWindowMenu}
\]

The meaning of these attributes are:
ProcessActivate and ProcessDeactivate

These two attributes correspond closely to the WindowActivate and Window-Deactivate attributes. Recall that keyboard and mouse input is always directed to the active window. The parent interactive process that contains this window is the active process. If the input focus is moved to a window that is not owned by the interactive process then the ProcessDeactivate attribute function is evaluated to inform the program that the interactive process has become inactive. If an inactive process obtains the input focus, its ProcessActivate attribute function is evaluated to inform the program that it has become active again.

ProcessClose

This attribute corresponds closely to the WindowClose attribute. If for some reason the interactive process is requested to be closed, its ProcessClose attribute function is evaluated. It can take the opportunity to save data to disk and to ask the user if the process can be closed safely. It is however the responsibility of the program to terminate the process. It is good programming practice to always include this attribute because the user of your program expects to be able to close your application in this way.

ProcessOpenFiles

Associating this attribute to an interactive process means that the interactive process can respond to user requests to open a number of files. The callback function has the somewhat verbose type ProcessOpenFilesFunction:

:: ProcessOpenFilesFunction st ::= [String] -> st -> st

The [String] argument contains the full pathnames of the files to be opened. It is good programming practice to include this attribute if your application allows users to open files because it will let your application behave in the platform conform way.

ProcessWindowPos, ProcessWindowSize, and ProcessWindowResize

In the current implementation these attributes have no effect. When they will, it will be the following: the process window is the root window in which all top-level user interface elements are created of an interactive process are located. On a Macintosh the process window is simply the screen. On the Windows platform it is also the screen in case of NDI processes. For SDI and MDI processes it is a window frame that can be resized by the user. For the latter case, the -Pos and -Size attributes specify the initial position and size in terms of the screen. In addition, if the user resizes the process window (by actually resizing the window or by changing the screen resolution), and a ProcessWindowResize attribute has been specified, then a function of type ProcessWindowResizeFunction is evaluated:

:: ProcessWindowResizeFunction st ::= Size -> Size -> st -> st

The first Size argument is the size of the process window before the resize occurred, and the second argument is the size afterwards.

ProcessToolbar

The process toolbar is a combination of menus and controls. The purpose of the toolbar is to present to the user of your application a list of the most frequently used commands, also called tools in this context. These commands are
supposed to be visualised via easily recognisable icons, currently represented by means of bitmaps. An optional string argument provides a brief textual description of the tool, in a similar way as the ControlTip attributes do (discussed in Section 7.1). It is also possible to include a separation between groups of related tools by using the ToolbarSeparator alternative.

:: ToolbarItem st
   = ToolbarItem Bitmap (Maybe String) (IdFun st)
   | ToolbarSeparator

The current implementation does not support enabling and disabling of tools, nor does it allow the callback function to receive modifier keys.

**ProcessNoWindowMenu**

This attribute is valid only for MDI processes. In the default case every MDI process has a special Windows menu (discussed in Section 8.3). If this attribute is set, then this menu is not added to the menu system of the interactive process.

### 11.2 Interactive process creation

The basic function to create individual processes from a World environment is `startIO`, which we have encountered many times in the preceding chapters. For completeness we repeat its type definition:

```haskell
startIO :: !DocumentInterface
   . l
   !(ProcessInit (PSt . l))
   ![ProcessAttribute (PSt . l)]
   !*World
   -> *World
```

Its arguments are identical to the arguments of the `Process` type constructor given in Section 11.1. The function `startIO` creates an interactive process with the same arguments and terminates only when this process and its child processes have terminated. It is a specialised function which uses the more general overloaded function `startProcesses` of the type constructor class `Processes`.

```haskell
class Processes pdef where
   startProcesses :: !pdef ![pdef !*World -> *World
   openProcesses :: !pdef ![pdef !(PSt . l) -> PSt . l
```

The type constructor `Process` is of course an instance of this class. The actual implementation of `startIO` is nothing but a shorthand for this instance:

```haskell
startIO :: !DocumentInterface . l !(ProcessInit (PSt . l)) ![ProcessAttribute (PSt . l)]
   !*World -> *World
startIO documentInterface local init atts world = startProcesses (Process documentInterface local
```
The other Processes member function openProcesses also creates an interactive process, but now in the context of an interactive process itself. Interactive processes can not use startIO because the World environment is not available from the PSt environment. When applied to an interactive process definition, openProcesses adds an initial version of that interactive process to the process state administration. At some point in time that interactive process will be initialised and joins the game. So process creation is asynchronous.

In addition to the Process type constructor instance, you can also create a list of interactive processes using the above Processes class member functions. This is stated concisely with the following type class instance declaration in module Std-Process:

instance Processes [pdef] | Processes pdef

So, if $p_1 \ldots p_n$ are Processes instances, then the expression (openProcesses $\{p_1 \ldots p_n\}$) is a process state transition function that creates these interactive process instances.

There is no special parent-child relationship between an interactive process and the interactive processes that it creates. For instance, termination of one interactive process (using closeProcess) has no consequence for the other processes. Indeed, the only entity that can terminate an interactive process is the interactive process itself (or, using brute force from the Operating System). This is one of the reasons why it is good programming practice to always add the ProcessClose attribute to every of your process definitions to allow graceful termination of your program.

11.3 Examples

In this section we give some examples of the use of interactive processes.

11.3.1 Talk revisited

In this example we have a new look at the talk example of Section 10.4.1. In that version, the program created one interactive process, using startIO, which opened the two talk windows. In the new version instead of a talk window we create a SDI process. Receivers are still used to send the user typed messages to each of the talk windows. The menu is now created for both processes. Below we discuss the differences. Figure 11.1 shows the program in action.

The initialisation of the new talk program is of course different. Instead of the function openTalkWindow that opened a talk window we now define a function talk that defines a talk process which contains the talk window, menu, and receiver. This implies that the RId values have to be created beforehand. We obtain the following Start rule:

Start :: *World -> *World
The menu definition is now moved inside the talk process. It is almost identical to the old version. The only difference is termination of the application. In the old version termination was no issue because there was only one interactive process with two talk windows. In the new version, closing one talk process won't close the other. Instead, before closing its parent process, the quit function sends a new message alternative Quit to the other process to request termination. So the message type is changed into:

:: Message = NewLine String | Quit

and, accordingly, the menu function quit into:

quit :: (RId Message) (PSt .1) -> PSt .1
quit you pst = closeProcess (snd (syncSend you Quit pst))

Of course the definition of the receiver function receive now handles two alternatives. When receiving (NewLine text), it proceeds as before, and when it receives Quit it only needs to terminate its parent process, knowing that the requesting process will terminate itself.

receive :: Id Message (PSt .1) -> PSt .1
receive outId (NewLine text) pst=:{io}
    = {pst & io=setEditControlCursor outId (size text) (}
These are the major differences. Here is the complete program.

```haskell
module talk

// ******************************************************************************
// Clean tutorial example program.
//
// This program creates two interactive processes that communicate via message
// passing.
// In a future distributed version this program can be used as a graphical talk
// application.
//******************************************************************************

import StdEnv, StdIO

:: Message    // The message type:
  = NewLine String    // transmit a line of text
  | Quit              // request termination

Start :: *World -> *World
Start world
  # (a, world) = openId world
  # (b, world) = openId world
  # (talkA,world) = talk "A" a b world
  # (talkB,world) = talk "B" b a world
  = startProcesses [talkA,talkB] world

talk :: String (RId Message) (RId Message) *env -> (Process,*env) | Ids env
  talk name me you env
    # (outId,env) = openId env
    # (inId, env) = openId env
    # input   = EditControl "" (PixelWidth (hmm 30.0)) 5
      [ ControlId inId
        , ControlKeyboard inputfilter Able
          (noLSi (input inId you))
        , ControlResize resizeHalfHeight
          ]
    # output  = EditControl "" (PixelWidth (hmm 30.0)) 5
      [ ControlId outId
        , ControlPos (BelowPrev,zero)
        , ControlSelectState Unable
        , ControlResize resizeHalfHeight
          ]
    # receiver = Receiver me (noLSi (receive outId)) []
    # talkwindow = Window "Talk "++name" (input:+:output:+:receiver)
      [ WindowViewSize (w=hmm 80.0,h=120) ]
    # menu = Menu "Talk "++name"
      [ MenuItem "&Quit"
        , MenuShortKey 'q',MenuFunction (noLS (quit you))
        ]
  = (Process SDI Void
    (snd o seqList [openWindow Void talkwindow.openMenu Void menu])
    [ProcessClose (quit you)]
    , env
  )

where
  inputfilter :: KeyboardState -> Bool
  inputfilter keystate
```

11.3. EXAMPLES
11.3.2 Clock revisited

In this example we are going to turn the clock example of Section 9.1.2 into a stopwatch component that can be added in an arbitrary interactive process. The stopwatch commands will be to reset timing, pause timing, continue timing, and close the stopwatch component. All stopwatch definitions are placed in the module stopwatch.icl. The function stopwatch defines the stopwatch component. A main module, usestopwatch.icl that opens and controls the stopwatch is also defined. This module defines an interactive process with a menu to control the stopwatch component. Figure 11.2 gives a snapshot of the program in action. We first look at the stopwatch and then at the main program.

![Figure 11.2: The stopwatch and control processes.](image)

The stopwatch component

The original clock program created an interactive process with three timers and a dialogue. Each of the three timers changes a local state that keeps track of the elapsed seconds, minutes, and hours. This situation is schematised in Figure 11.3. The stopwatch process is controlled by sending messages to a 'gateway' receiver. The timers are extended with a receiver component that handles the commands reset, pause, and continue. These commands will be sent to them by the gateway.
receiver. This gateway receiver handles the close command. Finally, the stopwatch process creates the same dialogue as the original clock program. The stopwatch process is schematised in Figure 11.4.

Figure 11.4: The structure of the stopwatch process.

So the new components are the gateway receiver and the receiver components of the timers. We first look at the gateway receiver and then timer receivers. Both receivers accept messages of the following type:

```
:: StopwatchCommands = Reset
    | Pause
    | Continue
    | Close
```

The alternatives of the algebraic type StopwatchCommands correspond of course with the stopwatch commands reset, pause, continue, and close. The gateway receiver function receive, on receiving the Close message simply terminates the interactive process by applying closeProcess to its process state. Every other message is routed to the timer receiver components which are identified by the list timerinfos. Here is the function definition of receive:

```
receive :: StopwatchCommands (PSt .1) -> PSt .1
receive Close pst
    = closeProcess pst
receive msg pst
    = snd (seqList [syncSend timerRId msg\{timerRId\}<timerinfos] pst)
```

The timer receiver components receive only the StopwatchMessage alternatives Reset, Pause, and Continue. In the clock example, the timer was parameterised with its timer interval. In this example, we need to add parameters to identify the timer (because it is going to be reset) and its receiver component (because receivers must have an identification value). So the definition of a stopwatch timer now is:
The `reset` command should set the timer back to zero. One might suppose that it is sufficient to change only the value of the local state to zero, but that is not completely true. Resetting the stopwatch can occur at any moment. At that moment the timer should be synchronised with its local state. This can be done by first disabling and then enabling the timer. For this purpose the `StdTimer` functions `disableTimer` and `enableTimer` should be used. The reason that it works is because `enableTimer`, when applied to a disabled timer, synchronises the timer with the moment of evaluation. It does nothing in case the indicated timer was already enabled. Finally, on receiving the `Reset` message, the timer receiver component must set the corresponding text field of the dialogue to zero. This gives the following definition of the `Reset` alternative.

```plaintext
receive Reset (time, pst=: {io})
  # io = disableTimer timerId io
  # io = enableTimer timerId io
  # io = setControlText textId "00" io
  = (0, {pst & io=io})
```

The `pause` command should halt the timer until further notice (either `reset` or `continue`). This is easily done by disabling the timer:

```plaintext
receive Pause (time, pst=: {io})
  = (time, {pst & io=disableTimer timerId io})
```

The `continue` command should let the timer continue from where it was paused. This is easily done by enabling the timer:

```plaintext
receive Continue (time, pst=: {io})
  = (time, {pst & io=enableTimer timerId io})
```

The final details of the stopwatch component are to create the proper identification values for the timers and their receiver components, and to export its definition as an interactive process. This is done by the function `stopwatch`. The stopwatch process is defined as a process group with no interesting local or public process state (using the ubiquitous `Void` singleton type constructor). Its initialisation action first creates the required IDs for the dialogue, and then the parameters required for the timers. Then initialisation proceeds as described above.

```plaintext
stopwatch :: (RId StopwatchCommands) -> Process
stopwatch rid
  = Process NDI Void initialise' []
where
  initialise' pst
    # (dialogIds, pst) = accPIO openDialogIds pst
    # (timerInfos, pst) = accPIO openTimerInfos pst
    = initialise rid dialogIds timerInfos pst
```
For completeness, the definition module and implementation module of the stopwatch component are given below.

```
definition module stopwatch

// This module exports the types and functions needed to incorporate a stopwatch component.

import StdIO

:: StopwatchCommands
   = Reset
     | Pause
     | Continue
     | Close

stopwatch :: (RId StopwatchCommands) -> Process

implementation module stopwatch

// This program defines a stopwatch process component.
// It uses three timers to track the seconds, minutes, and hours separately.
// Message passing is used to reset, pause, and continue timing.
// The current time is displayed using a dialogue.

import StdEnv,StdIO

:: DialogIds
   = { secondsId :: Id
        , minutesId :: Id
        , hoursId :: Id
     }
:: TimerInfo
   = { timerId :: Id
       , timerId :: RId StopwatchCommands
       , timerInterval :: TimerInterval
    }
:: StopwatchCommands
   = Reset
     | Pause
     | Continue
     | Close

second := ticksPerSecond
minute := 60*second
hour := 60*minute

openDialogIds :: *env -> (DialogIds, *env) | Ids env
openDialogIds env
   # ( [secondsId,minutesId,hoursId:_ ] ,_ ) = openIds 3 env
   = ( { secondsId=secondsId,minutesId=minutesId,hoursId=hoursId }, env )

openTimerInfos :: *env -> ([TimerInfo], *env) | Ids env
openTimerInfos env
   # (tids,_env) = openIds 3 env
   # (rIds,_env) = openRIds 3 env
   # intervals = [second,minute,hour]
   = ( [ {timerId=tid,timerId=rId,timerInterval=i} \ 
          | tid< tids & rid<rIds & i<intervals
```
CHAPTER 11. INTERACTIVE PROCESSES

stopwatch :: (RId StopwatchCommands) -> Process
stopwatch rid
  = Process ND Void initialise' []
where
  initialise' pst
    # (dialogIds, pst) = accPId openDialogIds pst
    # (timerInfos, pst) = accPId openTimerInfos pst
    = initialise rid dialogIds timerInfos pst

initialise :: (RId StopwatchCommands) DialogIds [TimerInfo] (PSt .1) -> PSt .1
initialise rid {secondsId, minutesId, hoursId} timerInfos pst
  # (errors, pst) = seqList [ openTimer 0 (tdef timerinfo)
    \ timerinfo<timerInfos ] pst
| any ((<> NoError) errors
  = closeProcess pst
  # (error, pst) = openDialog Void ddef pst
| error<>NoError
  = closeProcess pst
  # (error, pst) = openReceiver Void rdef pst
| error<>NoError
  = closeProcess pst
| otherwise
  = pst
where
ddef {timerId,timerRId,timerInterval}
  = Timer timerInterval [Receiver timerRId receive []]
    [ TimerId timerId
    , TimerFunction tick
    ]
where
tick nrElapsed (time,pst<>({io})
    # time = (time+nrElapsed) mod maxunit
    # io = setControlText textId (toString time) io
    = (time, {pst & io<io})
receive Reset (time,pst<>({io})
  # io = disableTimer timerId io
  # io = enableTimer timerId io
  # io = setControlText textId "0" io
  = (0, {pst & io<io})
receive Pause (time,pst<>({io})
  = (time, {pst & io<disableTimer timerId io})
receive Continue (time,pst<>({io})
  = (time, {pst & io<enableTimer timerId io})

  (textId,maxunit) = if (timerInterval==second)
  (secondsId,60)
  (if (timerInterval==minute)
  (minutesId,60)
  (hoursId,24))

rdef = Dialog "Stopwatch"
  [ LayoutControl
    [ ListLS [ TextControl text [ControlPos (Left,zero)]
      \ text<["Hours: ","Minutes: ","Seconds: "]
    ]
  ]
  :: LayoutControl
  [ ListLS [ TextControl "0" [ControlPos (Left,zero)
    ,ControlId id
    ,ControlWidth (ContentWidth "00")
  ]
    \ id<[hoursId,minutesId,secondsId]
  ]
Using the stopwatch

Given the stopwatch component module `stopwatch`, we can create a program that
opens, uses, and closes the stopwatch. The program will be as simple as possible.
Its first action is to create the (RId StopwatchCommands) value, so that it can be
given conveniently to the initialisation function of the interactive process. As this
process will only have a menu, it is a simple document interface process.

Start :: *World -> *World
Start world
# (stopwatchid,world) = openRId world
  = startIO SDI Void (initialise stopwatchid) [] world

The initialisation function of the interactive process first creates the menu, `mdef`
and then the stopwatch.

initialise :: (RId StopwatchCommands) (PSt .1) -> PSt .1
initialise stopwatchid pst
# (error,pst) = openMenu Void mdef pst
| error<>NoError = closeProcess pst
| otherwise = openProcesses (stopwatch stopwatchid) pst

The menu triggers the stopwatch commands `reset`, `pause`, `continue`, and `close`. In
addition, a `quit` command is added to terminate the whole program.

mdef
  = Menu "&Stopwatch"
    ( MenuItem "&Reset"   [MemFunction (noLS (send Reset))])
    :+: MenuItem "&Pause" [MemFunction (noLS (send Pause))])
    :+: MenuItem "C&ontinue" [MemFunction (noLS (send Continue))])
    :+: MenuItem "&Close" [MemFunction (noLS (send Close))])
    :+: MenuSeparator []
    :+: MenuItem "&Quit" [MemFunction (noLS (closeProcess o
                        (send Close))))]
  ) []

Each of the stopwatch command menu functions is defined by the `send` function
which is parameterised with the corresponding StopwatchCommands message alternative.
Its purpose is to send its argument message to the gateway receiver of the
stopwatch process, identified by `stopwatchid`. If this fails it emits a system beep.
Here is its definition:

send msg pst
# (error,pst) = syncSend stopwatchid msg pst
| error<>SendOk = appPIO beep pst
| otherwise = pst
Here is the complete code of the main program.

```haskell
module usestopwatch

// ******************************************************************************
// Clean tutorial example program.
// This program creates a simple program that uses the stopwatch process.
// The program only has a menu to open the stopwatch and control it.
// ******************************************************************************

import StdEnv, StdIO
import stopwatch

Start :: World -> World
Start world
    # (stopwatchid,world) = openRId world
    = startIO SDI Void (initialise stopwatchid) [] world

initialise :: (RId StopwatchCommands) (PSt .1) -> PSt .1
initialise stopwatchid pst
    # (error,pst) = openMenu Void mdef pst
    | error<NoError= closeProcess pst
    | otherwise = openProcesses (stopwatch stopwatchid) pst
where
  mdef = Menu "Stopwatch"
    (: [MenuFunction (noLS (send Reset))]
    ,+:[MenuFunction (noLS (send Pause))]
    ,+:[MenuFunction (noLS (send Continue))]
    ,+:[MenuFunction (noLS (send Close))]
    ,+:[MenuSeparator []]
    ,+:[MenuFunction (noLS (closeProcess o send Close))])

send msg pst
    # (error,pst) = syncSend stopwatchid msg pst
    | error<SendOk = appPIO beep pst
    | otherwise = pst
```
Chapter 12

Clipboard handling

The clipboard is a simple communication metaphor that can be used between and within applications. An application can write some data to the clipboard (typically text or pictures) which can be read at a later point of time by the same or another application. This mechanism is supported by the functions in the definition module StdClipboard. At the moment only text can be handled. Because we intend to incorporate pictures as well in the near future the current version is set up in such a way that it can be extended upward compatibly. For this purpose an abstract data type, ClipboardItem, is defined. Two overloaded functions from the type constructor class Clipboard take care that data types can be converted to and from ClipboardItems:

```haskell
:: ClipboardItem

class Clipboard item where
  toClipboard :: !item -> ClipboardItem
  fromClipboard :: !ClipboardItem -> Maybe item

instance Clipboard {#Char#}
```

By convention applications should provide several ‘popular’ data formats for the same clipboard content in descending order of accuracy. For instance, a text processor can first store its private format for the laid out text including font and style information, followed by an ASCII version of the same text, followed by a picture of the laid out text. For this reason the function `setClipboard` that writes the clipboard is not applied to one single clipboard data item but a list of them. The previous content will be erased. Programs are supposed to provide only one data item of each format, so `setClipboard` removes all duplicate formats from this list. Note that providing `setClipboard` with an empty list erases the clipboard.

```haskell
setClipboard :: ![ClipboardItem] !(PSt .l) -> PSt .l
```

The function that reads the clipboard, `getClipboard`, returns the list of the current clipboard content in descending order of accuracy. With the conversion function `fromClipboard` an application can determine easily if some data item is present that it can handle.

```haskell
getClipboard :: !(PSt .l) -> ([ClipboardItem],!PSt .l)
```
Finally, because reading in a complete clipboard can be time-consuming or space-consum ing the function `clipboardHasChanged` is provided that checks whether the clipboard has been updated since the last time the program checked it.

\[
\text{clipboardHasChanged} :: (!\text{PSt} .1) \to (!\text{Bool},!\text{PSt} .1)
\]

### 12.1 Example: a clipboard editor

To illustrate the use of the clipboard, we construct a small program that shows the current content of the clipboard and that can write some text to the clipboard (see Figure 12.1). It will create only one dialogue with two text fields. In the first text field, the `show` field, the content of the clipboard can be loaded by pressing the button to its right. In the second text field, the `set` field, the content of the clipboard can be stored by pressing the button to its right.

![Clipboard Viewer](clipboard.png)

**Figure 12.1: The clipboard program.**

The `show` text field and its activating button can be defined as follows:

```hs
showclip
 = EditControl "" width nrlines [ControlSelectState Unable
 ,ControlId showid
 ,ControlPos (Left,zero)
 ]

    :+: ButtonControl "Show" [ControlFunction (noLS show)]
```

The `show` text field is an edit text control that will not respond to keyboard input (because its `SelectState` attribute is `Unable`). It is identified by some Id of value `showid`. It will have some width and a height defined by the number of lines `nrlines`. The “Show” button, when selected, must read the content of the clipboard and figure out if there was a text clipboard item. It then sets the text of the `show` text field to the loaded clipboard content (an empty string if nothing was found). This action can be defined as follows:

```hs
show pst
    # (content,pst) = getClipboard pst
```
# text = getString content
= appPIO (setControlText showid text) pst

getString [clip:clips]
| isNothing item = getString clips
| otherwise = fromJust item

where
item = fromClipboard clip
getString []
= ""

The set text field and its activating button are defined as follows:

setclip
= EditControl "" width nrlines [ControlId setid
 ,ControlPos (Left,zero)
 ]
 :
:+: ButtonControl "Set" [ControlFunction (noLS set)]

The set text field is an edit text control which accepts keyboard input. It is identified by some Id of value setid. It has the same dimensions as the show text control. The “Set” button, when selected, must get the content of the set text control and write this to the clipboard. This action is defined as follows:

set pst
# (wst,pst)= accPIO (getParentWindow showid) pst
# text = fromJust (snd (getControlText setid (fromJust wst)))
= setClipboard [toClipboard text] pst

The definition of the clipboard viewing dialogue simply summaries these elements and adds a “Quit” button to terminate the program:

clipview = Dialog "Clipboard Viewer"
 (showclip :+: setclip :+: quit) []
quit = ButtonControl "Quit"
 [ControlFunction (noLS closeProcess)
 ,ControlPos (Center,zero)
 ]

The last details that remain to be defined are the opening of the interactive program. Here is the complete program code.

module clipboardview

// ******************************************************************************
// Clean tutorial example program.
// // This program creates a dialogue to display and change the current content
// of the clipboard
// ******************************************************************************

import StdEnv, StdIO
Start :: *World -> *World
Start world
  # (showid,world) = openId world
  # (setid,world) = openId world
  - startProcesses (Process NDI Void (initialise showid setid) []) world

initialise showid setid pst
  # (error,pst) = openDialog Void clipview pst
  | error<|NoError= closeProcess pst
  | otherwise = pst

where
  clipview= Dialog "Clipboard Viewer" ( showclip :: setclip :: quit ) []
  showclip= EditControl "" width nrlines [ ControlSelectState Unable
          , ControlId showid
          , ControlPos (Left,zero) ]
    :: ButtonControl "Show"
    [ ControlFunction (noLS show) ]
  setclip= EditControl "" width nrlines [ ControlId setid
          , ControlPos (Left,zero) ]
    :: ButtonControl "Set"
    [ ControlFunction (noLS set) ]
  quit = ButtonControl "Quit"
    [ ControlFunction (noLS closeProcess)
          , ControlPos (Center,zero) ]
    width = PixelWidth (hmw 50.0)
    nrlines = 4

show pst
  # (content,pst) = getClipboard pst
  # text = getString content
  - appPIO (setControlText showid text) pst

set pst
  # (wst,pst) = accPIO (getParentWindow showid) pst
  # text = fromJust (snd (getControlText setid (fromJust wst)))
  - setClipboard [toClipboard text] pst

getString [clip:clips]
  | isNothing item= getString clips
  | otherwise = fromJust item

where
  item = fromClipboard clip

getString []
  = ""
Chapter 13

Printing

In this chapter we introduce two modules that enable Clean programs to print: StdPrint (Appendix A.25) and StdPrintText (Appendix A.26). At first we will discuss the dialogues the user is confronted with when he wants to print something (Section 13.1). The StdPrint module can be used to print any arbitrary output. It is illustrated in the Sections 13.2 to 13.5. The StdPrintText module can be used when only text should be printed. Its use is explained in Section 13.6.

13.1 The user interface for printing

When a user wants to get some printed output, he has to specify several parameters for printing. These parameters can be divided into two groups: the print setup parameters determine the printer and the paper format that are used and the page orientation (portrait or landscape). The print job parameters include the number of copies and the range of pages to print. According to these two sets there are two dialogues to specify these parameters and two Clean types for objects that store this information. The dialogues are the print setup dialogue (Figure 13.1) and the print job dialogue (Figure 13.2). The types are the abstract data type PrintSetup and the JobInfo record (see Section 13.2). The print job dialogue launches the print job if the user does not cancel. In most applications the user has to answer this dialogue just before printing, but using the print setup dialogue is optional.

There is always a default PrintSetup which is returned by the defaultPrintSetup function. Such a PrintSetup can be edited with the print setup dialogue when it is applied to the printSetupDialog function. These functions have the following signatures:

:: PrintSetup

defaultPrintSetup :: !*env -> (!PrintSetup, !*env)
| FileEnv env
printSetupDialog :: !PrintSetup
| !env -> (!PrintSetup, !*env)
| PrintEnvironments env

The env parameter can be the World, Files, or the PSt. For the defaultPrintSetup function the env parameter can also be the I0St.

On Macintosh computers the print setup dialogue and the print job dialogue are
typically reachable from two different menu items. On Windows(95/NT) the print setup dialogue can usually only be reached by clicking on a button in the print job dialogue. The StdSystem module function printSetupTypical can be used to determine whether a separate menu item for the print setup dialogue is typical on the used platform.

### 13.2 The print function

The `print` function in the StdPrint module has the following signature:

\[
\text{print} :: \text{!Bool} \rightarrow \text{!Bool} \\
\hspace{1em} \hspace{1em} \text{PrintInfo} \rightarrow ([\text{IdFun} \rightarrow \text{Picture}], \text{Picture}) \\
\hspace{1em} \hspace{1em} \text{!PrintSetup} \rightarrow (\text{!PrintSetup}, \text{!env}) \\
\hspace{1em} \hspace{1em} \text{!PrintEnvironments} \rightarrow \text{env}
\]

\::\text{PrintInfo} \\
\hspace{1em} \hspace{1em} = \{ \text{printSetup} :\text{PrintSetup} \\
\hspace{2em} , \text{jobInfo} :\text{JobInfo} \\
\hspace{1em} \}

\::\text{JobInfo} \\
\hspace{1em} \hspace{1em} = \{ \text{range} :\!(\text{Int}, \text{Int}) \\
\hspace{2em} , \text{copies} :\!\text{Int} \\
\hspace{1em} \}

The parameters of \((\text{print doDialog emulateScreen pages printSetup env})\) have the following meaning:

\text{doDialog}
If True, the print job dialogue will pop up, otherwise printing will happen in the default way.

emulateScreen
If True, the screen resolution will be emulated, see Section 13.3.

pages
This function defines the printed output. It generates a list of drawing functions. Each drawing function corresponds to one page. The function can access a PrintInfo record and a Picture. The Picture parameter can be used to retrieve information like font metrics or string widths from a printer Picture that will have the same properties as the Picture environment used for printing the pages.

printSetup
The print setup. As mentioned before, the user can change this print setup from the print job dialogue under Windows(95/NT).

eenv
A print environment can be Files, World or PSt. If the print environment is the PSt, then the print function can update the contents of windows during printing (to be more exact: between the rendering of two pages). This is important on the Windows(95/NT) platforms, since printing can take some time. On the Macintosh platform printing is a blocking operation, so no window updating is necessary. If the print environment is either Files or the World, then no window updating will happen, and, on the Windows(95/NT) platform, no cancel dialogue for printing will pop up.

The PrintInfo record contains information that will be needed for generating the list of drawing functions. The fields of the JobInfo part of this record have the following meaning:

range
This field contains the numbers of the first and last page which the user has chosen via the print dialogue. If the user has chosen “all pages”, then the first page will be 1, and the last page will be 9999.
copies
These are the number of copies to generate. This will not necessarily be equal
to the number of copies as specified in the print job dialogue. Some printer
drivers take care of producing the appropriate number of copies themselves.
In that case the value of this field will be 1.

The PrintInfo record also contains a printSetup field. This print setup will be
used for printing. The PrintSetup value can be passed to the getPageDimensions
function:

getPageDimensions :: !PrintSetup !Bool -> PageDimensions

:: PageDimensions
  = { page :: !Size,
      , margins :: !Rectangle
      , resolution :: !(!Int,'Int)
     }

The PageDimensions record fields have the following meaning:

page
This field contains the size of the drawable area of a sheet in pixels, see
Figure 13.3. The print function sets the origin of such a Picture to zero.
So any drawing in a rectangle from zero to \{x=page.w-1, y=page.h-1\} will
be visible on the printed paper.

margins
This field contains information about the size of a whole sheet of paper. This
is not only the area in which the printer can draw, but also the margins of
the paper. The x-axis and y-axis coordinate values of corner1 are negative,
while the x-axis and y-axis coordinate values of corner2 are greater than the
 corresponding values of the page field, see Figure 13.3.

resolution
This field contains the horizontal and vertical resolution of the printer in dpi
(dots per inch).
These measurements of a printer picture depend on whether the screen resolution is emulated or not. Only if the Boolean parameter of the `getPageDimensions` function is True, then the values for a printer picture that emulates the screen resolution are returned.

The `print` function returns the `PrintSetup` that is stored in the `PrintInfo` record. Here is an easy example program, `printExample1`, that uses the `print` function:

```clean
defaultPrintSetup world = snd (print True False pages defaultPS world)
where

  pages = PrintInfo *Picture -> ([IdFun *Picture],*Picture)

  pages (printSetup, jobInfo=range=[first,last], copies=) picture
  # (resolution=())
  = getPageDimensions printSetup False

  # twoPages =( drawAt { x=100, y=100 } "Hello Printer"
  . drawAt { x=100, y=100 } { "Horizontal Resolution: "
   ++ toString xRes
   ++ "dpi." }

  oneCopy = twoPages % (first-1,last-1)

  = ( flatten (repeatn copies oneCopy), picture)
```

Running `printExample1` will pop up the print job dialogue. Let’s assume that the user chooses to print only one copy of the first two pages. These values will be passed to the `pages` function, via its first argument of type `PrintInfo`. So the values of `first` and `copies` are one, and the value of `last` is two. The list of drawing functions, which is returned by the `pages` function is then equal to `twoPages`. It contains two drawing functions, so two pages will be printed. The first printed page will contain the text “Hello Printer”, and the second page will contain the printer resolution.

If the user chooses to print only the first page, then the value of `last` is also one. The `%` operator selects only the first element of `bothPages` and so only one page is printed. If the user chooses to print all pages, then the value of `last` is 9999. The `%` operator then selects both pages, which will subsequently be printed. The application of `repeatn` ensures that the right number of copies is generated.

In general, printing is done in the same way as drawing on the screen, namely by applying drawing functions on a `Picture` environment. In the rest of this chapter printer `Pictures` and screen `Pictures` are distinguished. There are two differences between these two kinds of `Pictures`:

1. Functions which highlight parts of a `Picture`, or perform a `XOR` operation, will
have no effect on a printer Picture.

2. Bitmapped fonts cannot be drawn on both kinds of Pictures with the same sizes. This is because the pixel resolutions of both output devices are, in general, different. TrueType fonts work well for both kinds.

13.3 Reusing drawing functions

In many cases an application should be able to draw a certain content on printer Pictures as well as on screen Pictures. Ideally, such a program should use the same code for both output devices. But a problem arises because in the object I/O library two different units of measurement are used for drawing, namely points and pixels (see also Chapter 5).

The physical extent and position of a graphical object on the paper or on the screen always depends on Int values, which are passed to a drawing function. These values are interpreted either as point values or as pixel values. Point values occur only when the size of a font is specified. In all other cases pixel values are used, e.g. in `drawAt {x=100,y=100}`.

A point is approximately 1/72 of an inch. A pixel has of course also a physical extent, which can be measured in inches. But unlike a point, the size of a pixel varies from output device to output device: the size of a printer pixel can be about 3 to 4 times smaller than the size of a screen pixel. This implies that the result of applying a drawing function on a printer Picture will result in smaller images compared with screen Pictures. Only when text is drawn, this shrinking does not happen, because the size of the used font is specified in points, a resolution independent specification of physical extent.

Consider for example the following function:

```haskell
myDrawfunction picture
    # picture = drawAt {x=20,y=80} string picture
    # picture = drawAt {x=18,y=68} box picture
    = picture
where
    string = "This text is boxed"
    box = {box_w=120,box_h=15}
```

could cause a screen to contain the following:

```
This text is boxed
```

but the printed output could look like this:

```
This text is boxed
```

The reason is that the upper code uses constants to express pixel values. There are two ways to deal with this problem: implicit and explicit scaling.

Implicit scaling

Setting the second Boolean argument of the `print` function to `True` enables implicit scaling. This causes printer Pictures to emulate the screen resolution. This will suppress the shrinking effect. This approach has the advantage that
it's very easy to write a function that can be used for drawing on both a
screen and printer Picture. But not everything can be printed very well by
using this option. For instance lines on the paper can not be thinner than
on the screen. Further problems arise when the printer resolution is not a
whole multiple of the screen resolution. This causes rounding errors in the
emulation process. If the ratio between printer and screen resolution is for
instance 3.5, then the emulation of line drawing with a pen size of one screen
pixel will result in a line on the paper that is only 3 printer pixels wide. The
following drawing function draws five hundred horizontal lines:

```plaintext
seq [drawAt {x=0,y=y} {vx=500,vy=0} \ \ y<-[0..500]]
```

On paper space appears between the lines, but not on the screen.

**Explicit scaling**

In most cases using implicit scaling is preferrable because it leads to simple
programs. But if the disadvantages of implicit scaling are not acceptable,
the application has to perform the scaling itself. One way to obtain this is
to avoid constant pixel values. Consider for instance a program that draws a
string on a certain place in a window or sheet. The position of the string could
be specified by using values that are related to string widths or font metrics.
Since string width and font metrics always depend on a Picture, they will
reflect the resolution of the output device. The upper `myDrawfunction` on
page 162 could be rewritten to:

```plaintext
myDrawfunction picture
# (metrics,picture)= getPenFontMetrics picture
# (mwidth,descent) = (metrics.fMaxWidth,metrics.fDescent)
# height = fontLineHeight metrics
# (swidth,picture) = getPenFontStringWidth string picture
# box = {box_w=swidth*6*mwidth/5,box_h=0-height}
# picture = drawAt {x=20, y=80} string picture
# picture = drawAt {x=18-mwidth/5,y=80+descent} box picture
= picture
where
  string = "This text is boxed"
```

The constants in this example are not pixel values, because they are multiplied
with pixel values like height. This example also shows that writing resolution
independent drawing functions results in more complex code.

Another way to perform scaling is to use the printer and screen resolution
explicitly. Then the code fragment on page 162 could be rewritten to:

```plaintext
myDrawfunction (sclNom,sclDenom) picture
# picture = drawAt {x=scl 20,y=scl 80} string picture
# picture = drawAt {x=scl 18,y=scl 68} box picture
= picture
where
  string = "This text is boxed"
  box = {box_w=scl 120,box_h=scl 15}
  scl x = sclNom*x/sclDenom
```

For printing, `sclNom` should be the printer resolution, and `sclDenom` should
be the resolution of the screen. For drawing on the screen, both values should
be one. The StdPicture module function getResolution will return the horizontal and vertical resolution of a screen or printer picture:

\[
\text{getResolution} :: \text{!*Picture -> } (\text{!*Int,!*Int})\text{!*Picture}
\]

The accScreenPicture function (see Appendix A.30) can be handy in this context. The printer resolution can of course also be found in the PageDimensions record.

13.4 The printUpdateFunction function

The printUpdateFunction function in StdPrint is a good example of using implicit scaling. This function offers a very easy way to print the contents of a window. It has the following signature:

\[
\text{printUpdateFunction} :: \text{!*Bool }
\quad (\text{UpdateState -> *Picture -> *Picture})
\]
\[
\quad \text{[Rectangle]}
\quad \text{!*PrintSetup !*env}
\quad \text{-> (!*PrintSetup !*env)}
\quad \text{| PrintEnvironments env}
\]

The first and the last two parameters are identical to the first and last two parameters of the print function. Also the result has the same semantics as shown with the print function. Let's assume that an application has opened a window. The contents of the window is defined by its WindowLook attribute. This attribute contains a Look function which updates the contents of a window everytime this is required (see also Section 6.4.1). The Look is defined as:

\[
:: \text{Look ::= SelectState -> UpdateState -> *Picture -> *Picture}
\]

If such a Look function is curried with a SelectState, it can be passed to the printUpdateFunction function as the second argument.

The list of rectangles parameter specifies the parts of the view domain that should be printed. If such a rectangle is too big for one sheet of paper, it will be distributed on several pages.

13.5 The printPagePerPage function

The third function in the StdPrint module is the printPagePerPage function. The printUpdateFunction and the print functions are both specialisations of this function. This function performs a state transition function on a polymorphic state. The state transition function itself has to be passed to the printPagePerPage function. With each application of this state transition function the contents of the actually printed page is drawn. Furtheron this state transition function returns a Boolean value which indicates whether there are further pages to print. So it will be applied repeatedly, until this value becomes True.

The function printPagePerPage has the following signature (slightly changed for readability):
printPagePerPage
:: !Bool !Bool
  .unq
  (.unq PrintInfo *Picture -> ((Bool,Point2),(.state,*Picture)))
  (.state,*Picture) -> ((Bool,Point2),(.state,*Picture)))
!PrintSetup !*env
-> (Alternative .unq .state,!*env)
| PrintEnvironments env

:: Alternative unq state = Cancelled unq | StartedPrinting state

instance PrintEnvironments Files, World, (PSt .1)

The arguments of

printPagePerPage doDialog emulateScreen unq prepare stateTrans env

have the following meaning:

doDialog, emulateScreen, printSetup, and env
  Analogous to those in the print function.

unq
  If a unique object unq, e.g. a text file, has to be accessed and passed back
by printPagePerPage it should be passed via this parameter. If the user
 cancels printing via the print job dialogue, (Cancelled unq) will be returned.
Otherwise the prepare function is applied to this value.

prepare
  Calculates the initial state. In order to do so it can access the PrintInfo
record and a printer Picture. Drawing in that Picture will have no effect on
the printed output. The prepare also returns a Boolean and a Point2 value.
The Boolean value is False iff there is another page to print. The Point2
will be the origin of the next page, if there is one.

stateTrans
  This argument is the state transition function that generates the pages to
be printed. Just like the prepare function, each application of stateTrans
returns a Boolean and a Point2 value with the same meaning. If the user did
not cancel printing via the cancel dialogue, the printPagePerPage function
will return the final state in the StartedPrinting alternative constructor
of Alternative. Since this state is polymorphic, any arbitrary value can be
returned, e.g. the PrintInfo record.

13.6 Printing text

The module StdPrintText (Appendix A.26) offers functions to print text. These
functions are overloaded in the argument that specifies the text to print. In this
way the source of the text can be an arbitrary data structure, e.g. a text file or a
list of characters.

The printText1 function has the following signature:
printText1 :: !Bool !WrapMode !FontDef !Int !*charStream !PrintSetup !*env
-> ((!charStream,!PrintSetup),!*env)
|
class CharStreams charStream & PrintEnvironments env

A CharStreams instance contains the text to be printed. Its functionality is analogous to the behaviour of a text file. So a CharStreams value contains a finite sequence of characters and a position pointer that points to the actual character. Applying getChar retrieves this actual character and increases the position pointer. The Boolean return value is True iff this operation was successful. Characters can be read sequentially, until the eos (end of stream) function returns True. The actual position of a CharStreams value can be saved with the savePos function. The restorePos function resets the position pointer to the previously saved position.

The following example illustrates how to print a text that is stored in a list of characters. Therefore the CharStreams class is instanciated with a proper type ListCharStream.

module printCharList

// Clean tutorial example program.
// This program demonstrates the use of the function printText1.
It instantiates the CharStreams class with the ListCharStream type. Objects of type ListCharStream contain a list of characters, which should be printed.

```
import StdEnv, StdIO

:: <ListCharStream>
   = { list :: [Char], savedPos:: [Char] }

instance CharStreams ListCharStream where
   getChar sc=(list)
      = (not empty, if empty ' (hd list), {sc & list-if empty list (tl list)})
where
   empty = isEmpty list
   savePos sc
      = { sc & savedPos=sc.list }
   restorePos sc
      = { sc & list=sc.savedPos }
   eos sc=(list)
      = (isEmpty list,sc)

fontDef = { fName="Courier New", fStyles= [], fSize=9 }
```

Start world
```
# (defaultPS, world) = defaultPrintSetup world
= printText1 True NoWrap fontDef 4
   { list=['Hello again, printer'], savedPos= [] }
   defaultPS
   world
```

The StdPrintText module instanciates the CharStreams class with the *FileCharStream type. The following example illustrates how to print a text file, and how to use the functions fileToCharStream and charStreamToFile.

```
module printFile

// Clean tutorial example program.
// This program demonstrates printing a text file by using the function printText1.

import StdEnv, StdIO

fileName = "printFile.icl"
fontDef = { fName="Courier New", fStyles= [], fSize=9 }

Start world
```

The printText2 function allows you to print text with a header on each page. This function expects the same parameters as printText1, but takes in addition
two strings. The first string will appear on the left corner of each header, and the second string, concatenated with the current page number, will appear on the right side of each header. The previous example can be altered as follows:

```haskell
module printFileWithHeader

// **************************************************************
// Clean tutorial example program.
// This program demonstrates printing a text file by using the function printText2.
// **************************************************************

import StdEnv, StdIO

fileName = "printFileWithHeader.icl"
fontDef = { fName="Courier New", fStyles=[], fSize=9 }

Start world
  # [(ok, file, world)] = fopen fileName FReadData world
  | not ok
    = abort ("file " ++ fileName ++ " not found")
  # (defaultPS, world) = defaultPrintSetup world
  # [(charStream, _)] = printText2 "This file is printed with Clean" "page "
    True NoWrap fontDef 4
    [fileToCharStream file]
    defaultPS world
  # [(ok, world)] = fclose (charStreamToFile charStream) world
  | not ok
    = abort "can’t close file"
  | otherwise
    = world

Finally, the function printText3 permits full control over the look of a header and/or trailer (or footer) on each page. Its signature is:

```haskell
printText3 :: !Bool !WrapMode !FontDef !Int
    (PrintInfo *Picture -> (userInfo,(Int,Int),*Picture))
    (userInfo Int *Picture -> *Picture)
    !*charStream !PrintSetup !*env
    -> ([!*charStream,!*PrintSetup],!*env)
    | CharStreams charStream & PrintEnvironments printEnv
```

The fifth and sixth parameter of

```haskell
printText3 doDialog wrapMode fontParams spacesPerTab
textRangeFunc
eachPageDrawFunc
ccharStream printSetup env
```

have the following meaning:

```haskell
textRangeFunc
This function takes a PrintInfo record and a printer Picture and returns a triple. The first result can be any data of arbitrary type. This data will be passed by printText3 to eachPageDrawFunc. The second triple result is a pair (top, bottom), where top < bottom (printText3 aborts if this is not the case). The printed text will appear between these y-coordinates only, leaving you room to print a header and a trailer for each page.
```
eachPageDrawFunc

This function is responsible for drawing a header and trailer. As parameters it takes the data produced by textRangeFunc, the actual page number, and a Picture. This function will be called from printText3 for each page. It should return the Picture in which the header and/or trailer for the current page are drawn. Drawing is not restricted to any particular area on the printer Picture.
Chapter 14

TCP

In this chapter we cover Clean’s TCP interface. Blocking is an inherent problem with network I/O: a function blocks when it can not be reduced further until some external condition changes. A program can not do anything while a function blocks. An example of blocking is a program that tries to receive some data. Such a program could block until the data arrives.

It makes a difference whether a Clean program engages in event driven I/O or not. Programs engage in event driven I/O by evaluation of functions such as startIO or startProcesses (described in Chapter 11). Because the TCP interface can be used on the World environment (World programs) and the GUI environments FSt and IOST (GUI programs) we will distinguish between these kinds of programs in this chapter. For World programs blocking is not a real problem. This is not true for GUI programs: they should not block, or at least not block a “long” time. The “$\frac{1}{10}$ second rule” says that a GUI program should not take longer than $\frac{1}{10}$ second to process an event. Hence a GUI program should not block longer than this time (if the programmer respects that rule). The reason for this rule is that the user of an GUI program should have the impression that the program responds immediately to his actions. Furtheron, window update events should also be handled by the program promptly. Failing to do so can make your desktop windows look really ugly and uninformative.

After a short introduction to TCP in Section 14.1 some basic ideas are discussed in Section 14.2. Section 14.3 is about the blocking approach which is applicable in World programs. The non blocking approach for GUI programs is discussed in Section 14.4.

14.1 Introduction to TCP

TCP (Transmission Control Protocol) offers the possibility to transfer data between programs which are running on different computers via a network. TCP is a connection oriented protocol. Before data can be transferred a connection between two running programs has to be established. A TCP connection is always a duplex connection. This means that both sides can send and receive data when they are connected. After all data has been sent and received, the connection has to be teared down.

The Clean compiler produces sequential code. This is the reason why a program can not do anything else while it is blocking. If the Clean compiler would be able to produce non sequential code, blocking would be much less of a problem.
When establishing a connection between two programs, one of them plays the role of the server, and the other the client. The client has to know the address of the server. The address consists of two parts: an IP address and a port number. The IP address uniquely identifies every computer on the internet. An IP address is a 32 bit quantity, which is often written down in “dotted decimal form” like e.g. “131.174.33.11”. It is possible to establish different connections to one computer at the same time. The port number discriminates these connections. The port number is a 16 bit quantity. For some services some port numbers are reserved. For example, “HTTP” is reachable via the port number 80 and “telnet” via port number 23.

When a connection is established, the following happens: first the server has to listen on a certain port. Listening on a port means being prepared for receiving connection requests from clients. The client knows both this port number and the IP address of the computer, where the server is running. So he issues a connection request, which will reach the server via the internet. If the server accepts the connection request, the connection is established. Now data can be transferred. The sent data will reach the other side without being duplicated or altered in its chronological order.

A service called DNS (Dynamic Name System) helps us human beings to avoid having to remind IP addresses. The DNS service translates alphanumerical aliases such as “www.cs.kun.nl” into IP adresses.

14.2 Basic ideas

In Clean’s TCP interface channels play an important role. Intuitively a channel is like a pipe: what you put in on one end comes out on the other end after a while. We do not transport material goods like water or gas with our pipes, but information or rather messages. It is well possible that many messages reside within a channel at the same time while they are on their way to the other end. But it is not possible that they come out in another order than they were put in (we do not support out of band data). The most important things you can do with a channel are of course sending and receiving.

In our approach each channel type has a message type. This means, that we can only send and receive messages with that message type on a certain channel. But of course it is possible to define several channel types. For instance the following channel type is defined:

:: *TCP_RCharStream ::= TCP_RCharStream_ Char

The TCP_RCharStream type is a channel type with the message type Char. You don’t have to know how TCP_RCharStream is defined (note the underscore). In general a channel type is a type constructor with one argument, its message type. If you want to receive messages on such a channel you can apply the receive function:

receive :: !(*(ch . a) !*env -> !(.a,!*(ch . a),!*env)
            | ChannelEnv env & Receive ch

To help you understand this weird definition, let’s give an example how to specialise this type. It is necessary to know that TCP_CharStream is an instance of the Receive class, and that a ChannelEnv can be a World, IOST or PSt environment. The type of receive is defined as general as possible. The following function definition has a type that is more restrictive than it could be:
receive2 :: TCP_RCharStream *World -> (Char,TCP_RCharStream,*World)
receive2 ch env = receive ch env

The type of receive2 is obtained by uniform substitution of \(*\!(\!\text{char} .\!\! a)\!\) with TCP_RCharStream and \(!\!\text{char}\!\! a\) with Char. From the type of receive2 we see that we can receive a Char, if we apply the receive2 function on a TCP_RCharStream in a World. Channel types are uniquely attributed.

14.3 Blocking TCP in World programs

Since this section is about blocking use of TCP, the described methods are applicable for World programs. Section 14.4 discusses non blocking use of TCP in GUI programs.

The library modules that are relevant for this section are StdChannels, StdTCPDef and StdTCPChannels (see Appendices A.2, A.40 and A.39). The module StdTCP imports all these modules similar to the StdEnv and StdIO modules.

In general, a TCP session is divided into three phases: the connection establishment phase, the data transfer phase, and the disconnect phase. These three phases will be covered in the Sections 14.3.1 to 14.3.3. In these sections small example functions are described. These functions are combined to give an example for a client and a server program in Section 14.3.4. The technique of multiplexing is discussed in Section 14.3.5 which includes a fancy chat server example program. Section 14.3.6 introduces some further channels that can be used.

14.3.1 Establishing a connection

We show the process of establishing a connection first for client programs, and then for server programs.

The client program has to know the IP address and port number of the server program. There is a function that uses the DNS to get an IP Address:

lookupIPAddress :: !String !env -> (!Maybe IPAddress,!env)
| ChannelEnv env

The String that is passed can be an alphanumerical internet address such as martinpc.cs.kun.nl, but also a dotted decimal notation such as 131.174.33.11.

To establish a new connection, the client simply calls the connectTCP_MT function:

connectTCP_MT :: !Maybe !Timeout) !(IPAddress,Port) !env
-> (!TimeoutReport,!Maybe TCP_DuplexChannel,!*env)
| ChannelEnv env

:: Port := Int
:: Timeout := Int
:: TimeoutReport = TR_Expired
| TR_Success
| TR_NoSuccess

The optional Timeout value is measured in (platform dependent) ticks. The TimeoutReport informs about success or failure of the attempt to connect. If this value is TR_Success then Just a TCP_DuplexChannel is returned. A TCP_DuplexChannel is a record that contains two channels: one TCP channel to send and one to receive.
The TCP_SChannel and the TCP_RChannel types are channel types. The type of
the messages that can be received is the abstract data type ByteSeq. This type
resembles sequences of bytes. So with one message a sequence of bytes can be sent
or received (see Appendix A.40).

Example

We define a function that looks up an IP address and tries to connect to that
machine on port 2000. For simplicity we let the program abort if any of the
operations fails. If no failure occurs, the result is a TCP_DuplexChannel.

clientConnect :: !*World -> (!TCP_DuplexChannel,!*World)
clientConnect world
  # (mbIPAddr,world)= lookupIPAddress "martinpc.cs.kun.nl" world
  | isNothing mbIPAddr
  | = abort "DNS lookup failed"
  # ipAddress = fromJust mbIPAddr
  # (tReport,mbDuplex,world)
  | = connectTCP_MT Nothing (ipAddr,2000) world
  | tReport< TR_Success
  | = abort "can't connect to port 2000"
  | otherwise
  | # duplexChannel = fromJust mbDuplex
  | = (duplexChannel,world)

Now we discuss the process of establishing a connection for a server program. The
server has to call the openTCP_LISTENER function to listen on a given port:

openTCP_LISTENER :: !Port !*env
  -> (!OkBool,Maybe TCP_LISTENER,!*env)
  | ChannelEnv env

:: *TCP_LISTENER ::= TCP_LISTENER_ (IPAddress,TCP_DuplexChannel)
:: OkBool ::= Bool

The Port parameter should be the desired port number. If listening on the given
port succeeds, then the OkBool result will be True and the Maybe result will be Just
a TCP_LISTENER. If another program on the same machine already listens on the
given port, this attempt will fail. A TCP_LISTENER is a channel on which connection
requests from remote clients can be received. So the server does this by applying
the receive function on the TCP_LISTENER:

receive :: !(*(ch .a) !*env -> (!.a,!(*(ch .a),!*env)
  | ChannelEnv env & Receive ch
From the type definition of \texttt{TCP\_Listener} you can see that the message type \texttt{a} is a pair that consists of an \texttt{IPAddress} and a \texttt{TCP\_DuplexChannel}. Receiving a message of such a type establishes a new connection. The \texttt{IPAddress} is the IP address of the remote client that wants to connect. The \texttt{TCP\_DuplexChannel} is the new connection itself.

So a \texttt{TCP\_Listener} is a \emph{channel} on which \emph{channels} can be received. With one \texttt{receive} on a \texttt{TCP\_Listener} a \texttt{TCP\_SChannel} (to send) and a \texttt{TCP\_RChannel} (to receive) are obtained. (If you are familiar with the sockets \texttt{API} you will recognise that a \texttt{receive} on a \texttt{TCP\_Listener} resembles the \texttt{accept} call.)

\textbf{Example}

The following function listens on port 2000 and accepts one connection:

\begin{verbatim}
serverConnect :: !*World -> (!!TCP\_DuplexChannel,!*World)
serverConnect world
    # (ok,mbListener,world) = openTCP\_Listener 2000 world
    | not ok
        = abort "can't open Listener on port 2000"
    # listener
        = fromJust mbListener
    # ((_,duplexChannel),listener,world)
        = receive listener world
    # world
        = closeRChannel listener world
    = (duplexChannel,world)
\end{verbatim}

The \texttt{closeRChannel} function is discussed in Section 14.3.3.

14.3.2 Basic operations on channels

In the previous section we came in contact with three channel types: the \texttt{TCP\_Listener}, the \texttt{TCP\_SChannel}, and the \texttt{TCP\_RChannel}. These types are instances of type constructor classes that are defined in module \texttt{StdChannels} (Appendix A.2). The \texttt{TCP\_Listener} and the \texttt{TCP\_RChannel} are instances of the \texttt{Receive} class:

\begin{verbatim}
class Receive ch where
    receive\_MT :: (!Maybe !Timeout) -> (!TCP\_DuplexChannel,!*World)
    available :: (!TCP\_DuplexChannel,!*World)
    eom :: (!TCP\_DuplexChannel,!*World)

    receive\_MT : (!!ch .a) !*env
        | ChannelEnv env
        -> (!ch .a) !*env
    available : (!ch .a) !*env
        | ChannelEnv env
        -> (!ch .a) !*env
    eom : (!ch .a) !*env
        | ChannelEnv env
        -> (!ch .a) !*env

    ...\end{verbatim}

Each of these functions gets and returns a channel and an environment. "\texttt{MT}" is a shorthand for \texttt{maybe timeout}. The \texttt{receive\_MT} function tries to receive a message of type \texttt{a} on the channel. This function might block until the message arrives. A timeout value can be given to limit this time. The timeout value is given in (platform dependent) ticks. The returned \texttt{TimeoutReport} indicates whether the receive was successful (\texttt{TR\_Success}), whether the timeout expired (\texttt{TR\_Expired}), or whether the other side tear down the connection (\texttt{TR\_NoSuccess}). The \texttt{available} function
polls on the channel to check if a message is currently available. The eom (end of messages) function returns True if it is sure that available will never become True anymore. To understand these functions, the following model of receive channels is handy:

A receive channel consists of a message queue and is always in one of three states: the idle, available, or eom state, as shown in Figure 14.1.

![Diagram of a receive channel model](image)

Figure 14.1: Model of a receive channel

If the receive channel is in the available state, then there is a message in the message queue. If the channel is in the idle or eom state, then the message queue is empty. If a message arrives, then the state can change from idle to available, but if the channel is in the eom state, no message will arrive anymore. In this case the state does not change anymore. This happens when the other side tears down the connection.

To ease programming there are some specialisations of the functions above:

```haskell
receive :: !(ch . a) !*env -> (! . a, !(ch . a), !*env)  
  | ChannelEnv env & Receive ch
nreceive :: !Int !*!(ch . a) !*env -> (![.a], !(ch . a), !*env)  
  | ChannelEnv env & Receive ch
```

The receive function receives one message from the channel and (nreceive n) receives n messages. These functions are only partially defined. If the receive channel state is eom or becomes eom while the function blocks, then the program will abort! These functions should only be used if it is certain that the specified amount of data will be received.

Example

The following function receives a byte sequence and compares it with a given string. If the strings are different, the program aborts.

```haskell
serverReceive :: String TCP_RChannel *World  
  -> (TCP_RChannel, *World)
serverReceive expectedMessage rChannel world  
  # (message, rChannel, world) = receive rChannel world
  | toString message<>expectedMessage  
      = abort "received wrong message"  
  | otherwise  
      = (rChannel, world)
```
A special role is played by channels on which channels can be received, currently only the TCP Listener. If a message is available on such a channel it might be possible that a subsequent receive_M will return Nothing. This happens when a client tries to connect to a server, but disconnects again before the connection is accepted (received) from the server. Furtheron eom will never get True for this kind of channels.

On all other channels it holds that if available is True, receiving on that channel will not block and will return Just a message.

Before we discuss the basic operations on send channels, we introduce a model for these channels, see Figure 14.2.

\[\text{Figure 14.2: Model of a send channel}\]

A send channel has also three states: sendable, full, and disconnected. Furtheron there is a buffer of unsent bytes. If the channel is in the disconnected state, then no sending of data is possible. This happens when the remote side tears down the connection. The disconnected state is the final state (similar to the eom state for receive channels).

Sending on a channel can be blocking. Since this is not as obvious as in the receive case, here is a scenario where sending becomes blocking: let's assume a TCP connection, where data is sent only in one direction, from a "producer" program to a "consumer" program. Let's also assume that the producer sends much faster than the consumer, e.g. the producer sends one MByte per second but the consumer receives only one KByte per second, because he doesn't want to receive faster. It is obvious, that this does not work forever. At a certain point in time all buffers in the two participating computers and in the computers between them get full. The producer has to be stopped in his overproduction then, which means, that the producer's sending operation will block.

This explains why there are sendable and full states, and the message queue. If a channel is in the full state, then trying to send will not pump any data into the internet. Instead, the data will be queued locally in the channel's internal buffer. If the channel state changes from full to sendable then it's reasonable to flush this buffer. The buffer is a part of the Clean heap and does not have a limited size (apart from the usual memory limitations).

Now let's have a look at the basic operations on send channels:

class Send ch where  
  send_MT :: !((Maybe !Timeout) !.a !(*(!ch .a) !*env)  
          -> (!TimeoutReport, !Int, !(*(!ch .a), !*env)
The send function sends a message and the nsend function sends a list of messages. Both functions take a (Maybe Timeout) argument. If not all of the data could be sent within the timeout period, the rest will be queued in the send channels internal buffer. This internal buffer will not be sent automatically, it can be sent (flushed) with the flushBuffer function. This function tries to send as much as possible from the internal buffer. The Int value that is returned by these three functions is the number of sent bytes. If the timeout expires before everything could be sent, then the returned TimeoutReport will be TR_Expired. If the send channel’s state changed to disconnected during the send operation the TimeoutReport will be TR_NoSuccess. Only if all data was sent, this value will be TR_Success. The size of the internal buffer in bytes is returned by the bufferSize function. The disconnected function is used to poll whether the channel is in the disconnected state.

The following functions are provided to avoid having to specify timeouts for sending. They are specialisations of the functions in the Send class.

send :: ![a] !(*ch .a) !*env -> !((*ch .a),!*env) | ChannelEnv env & Send ch
nsend :: ![a] !(*ch .a) !*env -> !((*ch .a),!*env) | ChannelEnv env & nsend_MT ch

Example

The following function sends the message “hello server”.

clientSend :: TCP_SChannel *World -> (TCP_SChannel,*World)
clientSend sChannel world = send (toByteSeq "hello server") sChannel world

The MaxSize class allows to limit the size of the received byte sequences for receive channels. For further details see the StdChannels module.

Any object whose type is an instance of the toString class can be converted into a ByteSeq:

toByteSeq :: x -> ByteSeq | toString x

And vice versa: ByteSeqs can be converted into strings, because they are an instance of toString.
14.3.3 Tearing down a connection

Using TCP there are basically two ways to tear down a connection: *graceful* and *abortive* disconnects. The abortive disconnect happens when the *abortConnection* function is applied on a send channel. If the channel should be closed gracefully, then the *closeChannel_MT* function should be applied. Both functions are member of the *Send* class:

```haskell
class Send ch where
  closeChannel_MT :: !(Maybe !Timeout) !*(ch .a) !*env
  abortConnection :: !*(ch .a) !*env
```

The *closeChannel_MT* function takes a *(Maybe Timeout)* argument because it tries to send the channel's internal buffer before closing the channel. This function has also a specialized version without a timeout:

```haskell
closeChannel :: !*(ch .a) !*env -> !*env | ChannelEnv env & Send ch
```

The internal buffer will not be sent when the *abortConnection* function is used. Furthermore, it is possible that data that is already sent will not reach the other side, when this function is used.

To close receive channels the *closeRChannel* function should be used:

```haskell
class closeRChannel ch :: !*(ch .a) !*env -> !*env | ChannelEnv env
```

This function will not block.

14.3.4 Putting it together

The example functions that were introduced in Section 14.3.1 and 14.3.2 are now assembled together into two programs: *client* and *server*. The client attempts to connect to the server and sends the message "hello server". The server checks if the received message matches that text. Both sides close the connection gracefully afterwards. It is assumed that the server runs on a machine called *martinpc.cs.kun.nl*.

Here is the client program:

```haskell
module client

// *******************************************************
// Clean tutorial example program.
//
// This program implements a minimal TCP session (client).
// *******************************************************
import StdEnv, StdIO, StdTCP

clientConnect :: !*World -> (!TCP_DuplexChannel,!*World)
clientConnect world
  # (mbIPAddr,world) = lookupIPAddress "martinpc.cs.kun.nl" world
  | isNothing mbIPAddr
```
- abort "DNS lookup failed"
# ipAddr  = fromJust mbIPAddr
# (tReport,mbDuplex,world)  = connectTCP_MT Nothing (ipAddr,2000) world
| tReportSuccess
  = abort "can't connect to port 2000"
| otherwise
  # duplexChannel  = fromJust mbDuplex
  = (duplexChannel,world)
clientsend  :: TCP_SChannel *World  ->  (TCP_SChannel,*World)
clientsend  sChannel world  = send (toByteSeq "hello server") sChannel world

Start world
  # ((sChannel,rChannel),world)  = clientConnect world
  # (sChannel,world)  = clientSend sChannel world
  # world  = closeChannel sChannel world
  = world

And here is the server:

module server

// ***************************************************************
// Clean tutorial example program.
// //
// This program implements a minimal TCP session (server).
// ***************************************************************

import StdEnv, StdIO, StdTCP

serverConnect  :: *World  ->  (!TCP_DuplexChannel,*World)
serverConnect world
  # ((ok,mbListener,world))  = openTCP_Listener 2000 world
  | not ok
    - abort "can't open Listener on port 2000"
  | otherwise
    # listener  = fromJust mbListener
    # ([(_,duplexChannel),listener,world])  = receive listener world
    # world  = closeChannel listener world
    = (duplexChannel,world)

serverReceive  :: String TCP_RChannel *World  ->  (TCP_RChannel,*World)
serverReceive expectedMessage rChannel world
  # (message,rChannel,world)  = receive rChannel world
  | toString message>expectedMessage  = abort "received wrong message"
  | otherwise  = (rChannel,world)

Start world
  # ((sChannel,rChannel),world)  = serverConnect world
  # (rChannel,world)  = serverReceive "hello server" rChannel world
  # world  = closeChannel sChannel world
  # world  = closeRChannel rChannel world
  = world

14.3.5 Multiplexing

The `selectChannel_MT` function allows you to determine on which channel in a set of channels operations can be used in a non blocking way. This is very useful to determine that particular channel in a set for which data has arrived first. It’s signature is:

selectChannel_MT :: !(Maybe !Timeout)
14.3. BLOCKING TCP IN WORLD PROGRAMS

#: SelectResult
  = SR_Available
  | SR_EOM
  | SR_Sendable
  | SR_Disconnected

The \texttt{r\_channels} parameter can be a set of lists of receive channels. These lists can be combined with the \texttt{:^:} constructor (Appendix A.14). For instance, the following is defined:

#: *TCP\_RChannels = TCP\_RChannels [TCP\_RChannel]
#: *TCP\_Listeners = TCP\_Listeners [TCP\_Listener]

If \texttt{tcpRChs} is a list of \texttt{TCP\_RChannels} and \texttt{tcpLists} is a list of \texttt{TCP\_Listeners} then the following expression is a valid value for the \texttt{r\_channels} parameter:

\texttt{TCP\_RChannels tcpRChs :^: TCP\_Listeners tcpLists}

The \texttt{:^:} constructor allows us to combine lists of channels of different types. Similar sets of send channels can be created for the \texttt{s\_channels} argument. To specify an empty set of channels, it's also possible to pass \texttt{Void} as a value for \texttt{r\_channels} or \texttt{s\_channels}.

If the timeout expires, the function will return an empty list. Otherwise the \texttt{Int} part of each result pair will identify one of the channels out of \texttt{r\_channels} or \texttt{s\_channels} for which the \texttt{SelectResult} holds. Suppose for instance that both lists \texttt{tcpRChs} and \texttt{tcpLists} contain two elements. In the fragment:

```haskell
# r\_channels = TCP\_RChannels tcpRChs :^: TCP\_Listeners tcpLists
# (\(who,what\):[,r\_channels,\_world)
  = selectChannel\_MT Nothing r\_channels Void world
# (TCP\_RChannels tcpRChs :^: TCP\_Listeners tcpLists)
  = r\_channels
```

a set of four receive channels is passed via the \texttt{r\_channels} parameter to the select function. So the \texttt{who} value will be inbetween zero and three. Let's assume here for simplicity that the list that is returned by the \texttt{selectChannel\_MT} function only contains one element. If at first some data would have arrived for the first channel in the \texttt{tcpRChs} list, then \texttt{who} equals zero, for the other element of that list \texttt{who} equals one. Since the \texttt{TCP\_Listeners} data constructor is the right argument of \texttt{:^:}, the \texttt{who} value would be two or three respectively, if at first a connection request would have arrived for one of the listeners. In all these cases the value for \texttt{what} would have been \texttt{SR\_Available}. But if at first one of the \texttt{TCP\_RChannels} would get into the \texttt{conn} state, then the result would be \texttt{SR\_EOM}. Since only receive channels are passed to the \texttt{selectChannel\_MT} function, the \texttt{SelectResult} result can only be \texttt{SR\_Available} or \texttt{SR\_EOM}. The \texttt{SR\_Sendable} and the \texttt{SR\_Disconnected} values can only be returned if some send channels are part of the \texttt{s\_channels} parameter. As an example we pass a list \texttt{tcpSChs} of \texttt{TCP\_SendChannels} to the \texttt{selectChannel\_MT} function.
The \texttt{who} value will identify one of the send channels in the list (tcpSChs!!!\texttt{who}). Since no receive channels were passed to the \texttt{selectChannel\_MT} function, the \texttt{what} value can only be \texttt{SR\_Sendable} or \texttt{SR\_Disconnected}. These values indicate that the channel is in the \texttt{sendable} or \texttt{disconnected} state respectively.

Of course it is also possible to pass receive channels and send channels to \texttt{selectChannel\_MT}. The \texttt{SelectResult} result tells you whether a receive channel or a send channel is identified by the \texttt{Int} result. The numbering of channels starts with zero for receive channels and for send channels. If more than one channel could be selected by \texttt{selectChannel\_MT}, then priority is given from left to right.

As an example program we’ll discuss a server for a chat application. A chat client program (see Figure 14.3) will ask its user for a nickname and the address of the host where the server is running. After connecting to that server the client application will pop up a window with two edit controls. In the upper field the user can type some text which will be broadcasted via the server to all other people who are currently connected to the server. The client program is not a topic here, because it is a \textit{GUI} program, but it is incorporated in the examples part of this tutorial.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{chat.png}
\caption{Chatting via the internet}
\end{figure}

The server program first opens a listener on the port 2000. It uses a list of \texttt{ChanInfo} records. Each element of this list corresponds to one connection to a client. Apart from the two channels for the communication, the nickname for the connection is stored in a \texttt{ChanInfo} record. The set of receive channels that is passed to the \texttt{selectChannel\_MT} function consists of the listener and the receive channels for each open connection. There are three cases to handle: a new connection is made, data has been sent, a connection is closed.

\begin{verbatim}
module chatServer
  // ***************************************************************************
  // Clean tutorial example program.
\end{verbatim}
//
// This program demonstrates the usage of the selectChannel_MT function
// ******************************************************************************

import StdEnv, StdTCP, StdIO

chatPort := 2000

:: *ChanInfo
  = { sndChan :: TCP_SChannel
      , rcvChan :: TCP_RChannel
      , nickname :: String
    }

Start :: !World -> *World
Start world
# (ok, mbListener, world) = openTCP_Listener chatPort world
| not ok
| otherwise
  # (console, world) = stdout world
  - loop (fromJust mbListener) [] console world

loop :: !TCP_Listener !*ChanInfo !File !World -> *World
loop listener channels console world
# (sChans, rChans, nicknames) = unzip3 channels
# glue = TCP_Listeners [listener]
# TCP_RChannels rChans
# [[who, what], glue, ...] = selectChannel_MT Nothing glue Void world
# (TCP_Listeners [listener:_]) :: (TCP_RChannels rChans)
# channels = zip3 sChans rChans nicknames

| who==0
  // Case 1: someone wants to join the chatroom
  # (tReport, mbNewMember, listener, world)
  - receive_MT (Just 0) listener world
  | tReportOH_Success // The potential new member changed his mind
  - loop listener channels console world
  # (_, (sChannel, rChannel))
  - fromJust mbNewMember
  # (byteSeq, rChannel, world)
  - receive rChannel world
  # nickname = toString byteSeq
  # message = "*** " ++ nickname ++ " joined the group."
  # console = writes (message ++ \n"") console
  # channel = {sndChan=sChannel, rcvChan=rChannel, nickname=nickname}
  # channels = [channel, channels]
  # (channels, world) = broadcastString message channels [] world
  | nickname%(0, 3)="quit"
  - quit listener channels world
  // otherwise
  - loop listener channels console world

| what==SR_Available // Case 2: somebody has something to say
  # (channel, {rcvChan, nickname}, channels)
  - selectList (who=1) channels
  # (byteSeq, rcvChan, world)
  - receive rcvChan world
  # message = toString byteSeq
  # channels = channels++[(channel & rcvChan=rcvChan)]
  # (channels, world) = broadcastString (nickname ++ ": " ++ message)
  channel [] world
  - loop listener channels console world

| what==SR_EOM // Case 3: somebody leaves the group
  # (sndChan, rcvChan, nickname, channels)
- selectList (who-1) channels

# message
  "*** "++nickName"++" left the group"
# console
  - fwrites (message++, """) console
# (channels, world) = broadcastString message channels [] world
# world = seq (closeChannel sndChan, closeRChannel rcvChan) world
- loop listener channels console world

broadcastString :: !String ![ChanInfo] ![ChanInfo] !*World -> ([ChanInfo], !*World)
broadcastString string [] accu world
  - (reverse accu, world)
broadcastString string [channel=-(sndChan):channels] accu world
  # (sndChan, world) = send (toByteSeq string) sndChan world
  - broadcastString string channels [(channel & sndChan=sndChan):accu] world

selectList :: !Int -> (!.a, ![.a])
selectList n l
  # [left, [element:right]] = splitAt n l
  - (element, left++right)

quit listener channels world
- closeChannels channels (closeRChannel listener world)

closeChannels [] world
  = world
closeChannels [(sndChan, rcvChan): channels] world
  # world = closeChannel sndChan world
  # world = closeRChannel rcvChan world
  - closeChannels channels world

unzip3 :: ![ChanInfo] -> ([TCP_SChannel], ![TCP_RChannel], ![String])
unzip3 []
  = ([], [], [])
unzip3 [(sndChan, rcvChan, nickname): t]
  # (a, b, c) = unzip3 t
  - ([sndChan:a], [rcvChan:b], [nickname:c])

zip3 :: ![TCP_SChannel] ![TCP_RChannel] ![String] -> ![ChanInfo]
zip3 [] [] []
  = []
zip3 [sndChan:a] [rcvChan:b] [nickname:c]
  - ([sndChan=sndChan, rcvChan=rcvChan, nickname=nickname] : zip3 a b c)

Server processes typically run in the background. Macintosh users should be aware
that the MacOS was not designed as a multitasking operating system. Hence it is
not a default property of programs to be able to run in the background. But the
desired behaviour can be obtained if the ’Can Background’ bit of the ’SIZE’ resource
of the executable is set. The program that allows to edit these resources of files is
called ’ResEdit’.

### 14.3.6 More channels

Sometimes it is very handy if every single byte can be sent or received atomically.
For this purpose we implemented the following two channels:

:: *TCP_SCharStream ::= TCP_SCharStream_Char
:: *TCP_RCharStream ::= TCP_RCharStream_Char

On a TCP(S/R)CharStream characters can be sent (received). The functions
toSCharStream and toRCharStream convert TCP_S(R)Channels into TCP_S(R)Char-
Streams. The nsend and nreceive functions are handy to use with this kind of
channels. The following function takes a TCP_SChannel, converts it into a character
stream, and sends the characters “hello partner”.

s :: !TCP_SChannel !*World -> (!TCP_SCharStream, !*World)
s tcp_SChannel world
  # sCharStream = toSCharStream tcp_SChannel
  = nsend ['hello partner'] sCharStream world

If rCharStream is the corresponding TCP_RCharStream and world the World, then
the other side could receive these thirteen characters with the following function:

  nreceive 13 rCharStream world

Another kind of channels are the so called string channels. One handicap in using
raw TCP is that the ByteSeq packets are not atomic. That means, that sending
two byte sequences with sizes of for instance 10 and 14 bytes could be received on
the other side as one byte sequence with a size of 24 bytes. String channels use
their own protocol, which is built on top of the TCP protocol. If a string is sent on
a string channel, then at first a representation of the size of that string is sent, and
afterwards the contents. In this way it is possible to send empty strings as well as
strings with sizes of several megabytes. On the corresponding receive channel these
strings will be received as a whole. String channels should communicate only with
other string channels, since these channels use their very own protocol. It is possible
to use the setMaxSize function of the MaxSize class to limit the size of receivable
strings. In this way an application can be protected against bogus programs, which
claim to send strings that are too huge to fit in the memory of the used computer.
The definitions and instantiations for string channels can be found in the module
StdStringChannels (Appendix A36). We only show here the following:

:: *StringSChannel ::= StringSChannel, String
:: *StringRChannel ::= StringRChannel, String

The character streams and the string channels can be passed to the selectChan-
nel_MT function. The naming convention is that the data constructors for the
objects that are passed to the selectChannel_MT function end with an additional
"s". For example, it is possible to pass TCP_RCharStreams and StringRChannels
to the selectChannel_MT function:

selectChannel_MT
  Nothing
  (TCP_RCharStreams [ch0,ch1] ::= StringRChannels [ch2,ch3])
  Void
  world

14.4 Non Blocking TCP in GUI Programs

As said before, GUI programs should not block for a "long" time. To explain how
this condition can be solved we take a look at a very similar method of handling
input, namely keyboard input. In World programs we simply can write something
like:

  # (line, console) = freadline console
CHAPTER 14. TCP

The program would block until the user presses the "Enter" key. In a GUI program we should not use the `freadline` function to get keyboard input. We have seen in Section 6.7.1 that we should specify a callback function as a part of the `WindowKeyboard` attribute. When the user presses a key, the runtime system will look up this callback function (which is stored in the `PSt`) and apply it to the key code and the `PSt`. The callback function performs the necessary actions on the program state as a reaction on the user's keyboard input.

With TCP connections similar things happen. One obvious difference is that a callback function for TCP is not specified as a part of a `WindowAttribute`. Instead it is specified as a `ReceiverFunction` which is a part of a receiver definition (see Chapter 10). To receive data we have to open a receiver. If via a TCP connection some data arrives, the `ReceiverFunction` of a receiver will be applied to this data. But not only arrival of data will cause a `ReceiverFunction` to be applied. In general, if a channel's state changes, the runtime system will generate a certain event on which a `ReceiverFunction` will be applied. There are two events for receive channels:

:: ReceiveMsg m = Received m | EOM

and two events for send channels:

:: SendEvent = Sendable | Disconnected

A `(ReceiveMsg m)` event informs the application that the message `m` has arrived. The `EOM` event informs about closure of the channel. The two `SendEvents` inform the application that the state of a send channel changed to sendable or disconnected.

Another underlying idea is the following: when receivers are opened, receive channels are eaten but send channels are not eaten!

Eating has to do with the uniqueness typing system of Clean. A function eats an object if the type of that object is uniquely attributed, and not returned.

Example

Let's have a look at the following two functions:

```
sum_eating :: *(Int) -> Int
sum_eating {[0]=a0,[1]=a1} = a0+a1

sum_not_eating :: *(Int) -> (Int, *(Int))
sum_not_eating a=:{[0]=a0,[1]=a1} = (a0+a1, a)
```

`sum_eating` eats its unique array argument, but `sum_not_eating` doesn't. As a consequence, the array can not be used anymore if it is passed to `sum_eating`. The following application would be rejected by the type system:

```
Start = sum_eating a + a.[0]
where
  a = {47,11}
```

Value `a` is not unique on the right hand of the equal sign, because it is used twice. It was tried to use `a` in the expression `a.[0]`, although `sum_eating` has eaten `a`. Fortunately we can calculate the desired sum with the following rule:
Start
# (s1,a) = sum not eating a
= s1 + a.[0]
where
  a = \{47,11\}

As a consequence, it is impossible to apply any function to a receive channel, after a receiver was opened for such a channel.

Let’s examine, how we can open receivers for receive channels.

For each receive channel there is an algebraic receiver definition type that is used for opening a receiver. An object of such a type is passed to the openReceiver function, as shown in Chapter 10. For TCP_RChannels there is a TCP_RReceiver:

```
:: *TCP_RReceiver ls pst
   = TCP_RReceiver
     Id TCP_RChannel
     (ReceiverFunction (ReceiveMsg ByteSeq) *(ls,pst))
     [ReceiverAttribute *(ls,pst)]
```

```
:: ReceiverFunction m st ::= m -> st -> st
```

To open such a receiver, an Id, a TCP_RChannel, a ReceiverFunction, and ReceiverAttributes have to be specified. The Id can be used to disable or close the receiver. If data has arrived on the channel, the ReceiverFunction will be called with the Received alternative. If the connection is tear down by the remote peer, then the ReceiverFunction will be called with the EOM alternative. Similar definitions are the TCP_ListenerReceiver (for TCP_Listeners), the TCP_CharacterReceiver (for receiving character by character) and the StringChannelReceiver (for StringRChannels) (see Appendixes A.40 and A.36).

**Example**

We show a function \( f \) that opens a receiver for string channels. The ReceiverFunction \( \text{rcvFun} \) stores the received strings in a file which is also passed to \( f \). The file is put in the local state of the receiver. When the EOM event occurs, the receiver will be closed by the runtime system after evaluation of the EOM alternative. It is not necessary to close such a receiver explicitly.

```
f :: StringRChannel *File (PSt .1) -> PSt .1
f stringChannel file pst
  # (rId, pst) = openRId pst
  # (error,pst) = openReceiver
  file
    (StringChannelReceiver
     rId
     stringChannel rcvFun []
    ) pst
  | error<>NoError = abort "an error occurred"
  | otherwise = pst
where
  rcvFun (Received string) (file,pst)
    = (fwrite string file, pst)
  rcvFun EOM (file, pst)
    = (undef, snd (fclose file pst))
```
Now let's turn our attention to receivers for send channels. This kind of receiver is called SendNotifier. To open a SendNotifier we use the function openSendNotifier which does not eat its channel argument:

```haskell
openSendNotifier :: .ls !(SendNotifier *(ch .a) .ls (PSt .1))
                   !(PSt .1)
                -> (!ErrorReport, !(*(ch .a), !PSt .1))
                | accSChannel ch & Send ch
```

```haskell
:: SendNotifier sChannel ls pst
  = SendNotifier
    sChannel
    (ReceiverFunction SendEvent *(ls,pst))
    [ReceiverAttribute *(ls,pst)]
```

As you can see no Id is used. Indeed, an Id is not needed because the only thing we want to do with a SendNotifier is to close it. This is done automatically when we close the corresponding send channel.

Because the send channel will not be eaten, it has to be stored somewhere in the program state.

To send some data we can apply the following functions, which are defined in the StdChannels module. These functions simply call their MT counterpart with a timeout of zero ("NB" is a shorthand for "non blocking"):

```haskell
send NB :: .a !(*(ch .a) !env -> (!(*(ch .a),!*env))
         | ChannelEnv env & Send ch
flushBuffer NB :: !(*(ch .a) !env -> (!(*(ch .a),!*env))
                 | ChannelEnv env & Send ch
```

Typically a GUI program uses the sendNB function to send data in a non blocking way. If not all of the data can be sent immediately because the send channel's state changed to full, then sendNB stores the unsent data in the send channel's internal buffer. When the flow conditions permit sending, the runtime system applies the SendNotifier's ReceiverFunction on the Sendable event. The ReceiverFunction should then try to flush the internal buffer by using the flushBufferNB function.

As an example we will discuss a server program that accepts one connection and echoes the incoming data. To do this it opens two receivers in its initialisation function: for the receive channel a receiver that handles the incoming data, and for the send channel a send notifier. The send notifier allows the application to perform flow control. It should be possible that the remote side wants to receive the echoed data much slower than it wants to send. The echo server monitors the size of the internal buffer of the send channel. If this buffer size is greater than zero, then the receiver which receives the incoming data will be disabled. It will be enabled only when due to a Sendable event the buffer can be flushed again.

```haskell
module echoServer
```
import StdEnv, StdTCP, StdIO

echoPort := 7

:: State = { duplex:: TCP_DuplexChannel // The channel, eom :: Bool // EOM occurred on receive channel }

Start = *World -> *World
Start world
  # (_, mbListener, world) = openTCPListener echoPort world
  # ([],_,world) = receive (from Just mbListener) world
  world = closeChannel listener world
  = startIO NDI {duplex=duplex, eom=False} initialise [] world

  /* initialise - the function to initialise the PST. */
  initialise :: (PST State) -> PST State
  initialise pst=(ls=ls={duplex={rChannel, sChannel}}, io)
    # (tcpPcvId, io) = openId io
    # pst = {pst & ls={ls & duplex={rChannel=undef, sChannel=undef}},
          , io=io }

  // Open a receiver for the receive channel
  # (error1, pst) = openReceiver tcpPcvId
  // (TCP_Receiver tcpPcvId rChannel rcvFun []) pst
  // Open a receiver for the send channel
  # (error2, sChannel, pst) = openSendNotifier tcpPcvId
  // (SendNotifier sChannel sndFun []) pst
  | error1<>Error | error2<>Error
    = abort "error: can't open receiver"
  | otherwise
    = (pst & ls={ls & duplex={rChannel=undef, sChannel=undef}})

  /* rcvFun - the callback function for the receive channels receiver. */
  rcvFun : (ReceiveMsg ByteSeq) (Id,PST State) -> (Id,PST State)
  rcvFun (received byteSeq) (tcpPcvId,pst=(ls=ls={duplex={rChannel, sChannel}}, io))
    # (channel, io) = send_NB byteSeq sChannel io
    # (bufSize, sChannel) = bufferSize sChannel
    # ls = {ls & duplex={dc & sChannel},
    | bufSize=0
      = (tcpPcvId,(pst & ls=ls, io=io))
    | otherwise  // Disable this receiver if the send channel is full
      # = disableReceivers [tcpPcvId] io
      = (tcpPcvId,(pst & ls=ls, io=io))
  rcvFun EOM (tcpPcvId,pst=(ls=ls={duplex={rChannel, sChannel}}, io))
    # (bufSize, sChannel) = bufferSize sChannel
    # pst = {pst & ls={ls & duplex={dc & sChannel},
     , eom = True
    , io=io }

    | bufSize=0  // All data has been sent, so close program
      = (tcpPcvId,closeProcess (close pst))
    | otherwise
      = (tcpPcvId,pst)

  /* sndFun - the callback function for the send channels receiver. */
  sndFun : SendEvent (Id,PST State) -> (Id,PST State)
  sndFun Sendable (tcpPcvId,pst=(ls=ls={duplex={rChannel, eom}, io}))
    # (sChannel, io) = flushBuffer_NB sChannel io
    # (bufSize, sChannel) = bufferSize sChannel
    # pst = {pst & ls={ls & duplex={dc & sChannel, sChannel}}}
Is it forbidden to use blocking functions in GUI programs? This question arises because using blocking functions usually results in nicer programs. The answer is: of course it is not forbidden. The only problem that arises when using blocking functions is that the program will simply do nothing else while it blocks. In particular it will not update the contents of windows. The programmer has to estimate himself whether this is acceptable or not. Furthermore it depends on many factors whether the functions will block or not. For example there is no problem to send data in a blocking way to a program that is known to receive fast enough.
Appendix A

I/O library

A.1 StdBitmap

definition module StdBitmap

// *************************************************************************************************
// Clean Standard Object I/O library, version 1.2
//
// StdBitmap contains functions for reading bitmap files and drawing bitmaps.
// *************************************************************************************************

import StdMaybe
from StdFile import FileSystem
from osbitmap import Bitmap
import StdPicture
export FileSystem World

openBitmap :: ![#Char] ![env] -> (Maybe Bitmap ![env] | FileSystem env
/* openBitmap reads in a bitmap from file.
   The String argument must be the file name of the bitmap.
   If the bitmap could be read, then (Just bitmap) is returned, otherwise Nothing is returned. */

getBitmapSize :: !Bitmap -> Size
/* getBitmapSize returns the size of the given bitmap.
   In case the bitmap is the result of an erroneous openBitmap, then the size is zero. */

resizeBitmap :: !Size !Bitmap -> Bitmap
/* zooms or stretches a bitmap. The second argument is the size
   of the resulting bitmap */

instance Drawables Bitmap
/* draw bitmap
   draws the given bitmap with its left top at the current pen position.
drawAt pos bitmap
   draws the given bitmap with its left top at the given pen position.
undraw(At)
   equals unfill(At) the box \{box_w=w,box_h=h\} with \{w,h\} the size of the bitmap. */
A.2 StdChannels

definition module StdChannels

// Clean Standard Object I/O library, version 1.2

// StdChannels defines operations on channels

from StdMaybe import Maybe
from StdOverloaded import ==, toString
from tcp import ChannelEnv

instance ChannelEnv World

class Receive ch where

receive_MT :: !(Maybe !Timeout) !*(ch .a) !env
          -> (!TimeoutReport, !Maybe !.a/, !*env)
          | ChannelEnv env

receiveUpTo :: !Int !*(ch .a) !env
            -> (![.a], !*(ch .a), !env)
            | ChannelEnv env

available :: !(Bool, !*(ch .a), !env)
            -> (!Bool, !*(ch .a), !env)
            | ChannelEnv env

eom :: !(Bool, !*(ch .a), !env)
        -> (!Bool, !*(ch .a), !env)
        | ChannelEnv env

/* receive_MT
   tries to receive on a channel. This function will block until data can be
   received, eom becomes true, or the timeout expires.

receiveUpTo max ch env
   receives messages on a channel until available becomes False or max
   messages have been received.

available
   polls on a channel whether some data is ready to be received. If the
   returne Boolean is True, then a following receive_MT will not block and
   return TR_Success.

eom ("end of messages")
   polls on a channel whether data can’t be received anymore.
*/

class closeRChannel ch :: !*(ch .a) !env -> !*env | ChannelEnv env

// Closes the channel

class Send ch where

send_MT :: !(Maybe !Timeout) .a !*(ch .a) !env
           -> (!TimeoutReport, !Int, !*(ch .a), !env)
           | ChannelEnv env

nsend_MT :: !(Maybe !Timeout) ![.a] !*(ch .a) !env
              -> (!TimeoutReport, ![Int], !*(ch .a), !env)
              | ChannelEnv env

flushBuffer_MT :: !(Maybe !Timeout) !*(ch .a) !env
                   -> (!TimeoutReport, !Int, !*(ch .a), !env)
                   | ChannelEnv env

closeChannel_MT :: !(Maybe !Timeout) !*(ch .a) !env
A.2. STDCHANNELS

-> ([TimeoutReport, !Int, !*env])
  | ChannelEnv env
abortConnection :: !*(ch .a) !*env
  -> !*env
  | ChannelEnv env
disconnected :: !*(ch .a) !*env
  -> ([!Bool, !*(ch .a) !*env])
  | ChannelEnv env
bufferSize :: !*(ch .a)
  -> ([!Int, !*(ch .a)])

/* send_MT mbTimeout a ch env
adds the data a to the channels internal buffer and tries to send this
buffer.
nsend_MT mbTimeout l ch env
adds the data l to the channels internal buffer and tries to send this
buffer.
flushBuffer_MT
tries to send the channels internal buffer.
flushChannel_MT
first tries to send the channels internal buffer and then closes the
channel.
abortConnection
will cause an abortive disconnect (sent data can be lost).
disconnected
polls on a channel, whether data can’t be sent anymore. If the returned
Boolean is True, then a following send_MT will not block and return
TR_NoSuccess.
bufferSize
returns the size of the channels internal buffer in bytes.

The integer value that is returned by send_MT, nsend_MT, flushBuffer_MT, and
flushChannel_MT is the number of sent bytes.
*/

// miscellaneous

class MaxSize ch where
setMaxSize :: !Int !*(ch .a) -> !*(ch .a)
getMaxSize :: !*(ch .a) -> ([Int, !*(ch .a)])
clearMaxSize :: !*(ch .a) -> !*(ch .a)

/* Set, get, or clear the maximum size of the data that can be received
*/

:: DuplexChannel sChannel rChannel a
  = { sChannel:: sChannel a
      , rChannel:: rChannel a
    }

:: TimeoutReport
  = TR_Expired
     | TR_Success
     | TR_NoSuccess

:: Timeout
  ::= Int // timeout in ticks

:: ReceiveMsg m = Received m
  | EDM
  // Receiving "EDM" automatically closes the receiver
:: SendEvent = Sendable
  | Disconnected
  // Receiving "Disconnected" automatically closes the receiver

instance == TimeoutReport
instance toString TimeoutReport
APPENDIX A. I/O LIBRARY

// derived functions

nreceive_MT :: !(Maybe !Timeout) !Int !(ch .a) !*env
  -> (!TimeoutReport, ![a], !(ch .a), !*env)
  | Receive ch & ChannelEnv env

/* nreceive_MT mbTimeout n ch env
   tries to call receive_MT n times. If the result is (tReport, 1, ch2, env2),
   then the following holds:
   tReport = TR_Success <-> length l = n
   tReport = TR_NoSuccess -> length l < n
*/

/* The following two receive functions call their "_MT" counterpart with no
   timeout. If the data can't be received because eom became True the function will
   abort.
*/
receive :: !(ch .a) !*env
  -> ([a], ![ch .a], !*env)
  | ChannelEnv env & Receive ch
nreceive :: !Int !(ch .a) !*env
  -> ([a], ![ch .a], !*env)
  | ChannelEnv env & Receive ch

/* The following three send functions call their "_MT" counterpart with no timeout.
*/
send :: ![a] !(ch .a) !*env
  -> ![ch .a], !*env)
  | ChannelEnv env & Send ch
nsend :: ![a] !(ch .a) !*env
  -> ![ch .a], !*env)
  | ChannelEnv env & nsend_MT ch
closeChannel :: ![a] !(ch .a) !*env
  -> ![ch .a], !*env)
  | ChannelEnv env & Send ch

/* The following two send functions call their "_MT" counterpart with timeout = 0.
   "NB" is a shorthand for "non blocking"
*/
send_NB :: ![a] !(ch .a) !*env
  -> ![ch .a], !*env)
  | ChannelEnv env & Send ch
flushBuffer_NB :: ![a] !(ch .a) !*env
  -> ![ch .a], !*env)
  | ChannelEnv env & Send ch
A.3 StdClipboard

definition module StdClipboard

  // ******************************************************************************
  // Clean Standard I/O library, version 1.2
  //
  // StdClipboard specifies all functions on the clipboard.
  // ******************************************************************************

import StdMaybe
from iostate import PST, IOST

  // Clipboard data items:
  :: ClipboardItem

  class Clipboard item where
    toClipboard :: !item -> ClipboardItem
    fromClipboard :: !ClipboardItem -> Maybe item
  *
  toClipboard makes an item transferable to the clipboard.
  fromClipboard attempts to retrieve an item of the instance type from the clipboard item.
  If this fails, the result is Nothing, otherwise it is (Just item).
  */

instance Clipboard {#Char}

  // Access to the current content of the clipboard:

setClipboard :: ![ClipboardItem] !(PST .1) -> PST .1
getClipboard :: !(PST .1) -> ![ClipboardItem]!(PST .1)
  *
  setClipboard replaces the current content of the clipboard with the argument list.
  Of the list only the first occurrence of a ClipboardItem of the same type
  will be stored in the clipboard.
  Note that setClipboard [] erases the clipboard.
  getClipboard gets the current content of the clipboard without changing the content.
  */

clipboardHasChanged :: !(PST .1) -> ![Bool]!(PST .1)
  *
  clipboardHasChanged holds if the current content of the clipboard is different
  from the last access to the clipboard.
  */
A.4 StdControl

definition module StdControl

// Clean Standard Object I/O library, version 1.2
// StdControl specifies all control operations.
// ************

import StdControlClass

controlSize :: cdef .ls (PSt .l)) !Bool
! (Maybe (Int,Int)) (Maybe (Int,Int)) (Maybe (Int,Int))
! (PSt .l)
-> (lSize .PSt .l) ! Controls cdef
/* controlSize calculates the size of the given control definition as it would be
opened as an element of a window/dialog.
The Boolean argument determines whether a window (True) or a dialog (False) is
intended.
The Maybe arguments are the preferred horizontal margins, vertical margins, and
item spaces (see also the (Window/Control)(H/V)Margin and
(Window/Control)ItemSpace attributes). If Nothing is specified, their
default values with respect to the window/dialog are used.*/

/* Functions that change the set of controls in windows/dialogues. */

openControls :: !Id .ls (cdef .ls (PSt .l)) ! (PSt .l)
-> (!ErrorReport !PSt .l)
| Controls cdef
openCompoundControls :: !Id .ls (cdef .ls (PSt .l)) ! (PSt .l)
-> (!ErrorReport !PSt .l)
| Controls cdef
openPopUpControlItems :: !Id !Index ! [PopUpControlItem (PSt .l)]
! (IOSt .l) -> IOSt .l
/* openControls
adds the given controls argument to the indicated window or dialog.
openCompoundControls
adds the given controls argument to the indicated compound control.
openPopUpControlItems
adds the PopUpControlItems to the indicated PopUpControl behind the item at
the given index position (counted from 1).
The window/dialog is not resized.
These functions have no effect in case the indicated window/dialog/compound
control could not be found (ErrorUnknownObject) or if controls are opened with
duplicate Ids (ErrorIdsInUse).*/

closeControls :: !Id [Id] !Bool ! (IOSt .l) -> IOSt .l
/* closeControls removes the indicated controls (second argument) from the
indicated window (first argument) and recalculates the layout iff the Boolean
argument is True.*/

closeAllControls :: !Id !IOSt .l) -> IOSt .l
/* closeAllControls removes all controls from the indicated window. */

closePopUpControlItems :: !Id !Index] ! (IOSt .l) -> IOSt .l
/* closePopUpControlItems closes PopUpControlItems by their Index position of the
indicated PopUpControl.*/
A.4. STDCONTROL

If the currently checked element of a PopUpControl is closed, the first remaining element of that PopUpControl will be checked.

/*

setControlPos :: :Id ![Id.ItemPos] ![IOSt .1] -> ([Bool,]IOSt .1)
/* setControlPos changes the current layout position of the indicated controls to their new positions.

If there are relatively layout controls, then their layout also changes. The window is not resized.

The Boolean result is False iff the window is unknown.
*/

/*

Functions that change the state of controls.

Only those Id arguments that refer to controls within the same interactive process are used to change the corresponding controls.
*/

showControls :: ![Id] ![IOSt .1] -> IOSt .1
showControl :: :Id ![IOSt .1] -> IOSt .1
hideControls :: ![Id] ![IOSt .1] -> IOSt .1
hideControl :: :Id ![IOSt .1] -> IOSt .1
/* (show/hide)Control(s) makes the indicated control(s) visible/invisible.

Hiding a control overrides the visibility of its elements, which become invisible.

Showing a hidden control re-establishes the visibility state of its elements.
*/

enableControls :: ![Id] ![IOSt .1] -> IOSt .1
enableControl :: :Id ![IOSt .1] -> IOSt .1
disableControls :: ![Id] ![IOSt .1] -> IOSt .1
disableControl :: :Id ![IOSt .1] -> IOSt .1
/* (en/dis)ableControl(s) (en/dis)ables the indicated control(s).

Disabling a control overrides the SelectStates of its elements, which become unselectable.

Enabling a disabled control re-establishes the SelectStates of its elements.
*/

markCheckControlItem :: !Id ![Index] ![IOSt .1] -> IOSt .1
unmarkCheckControlItem :: ![Id ![Index] ![IOSt .1] -> IOSt .1
/* (unm/m)arkCheckControlItem unmarks/marks the indicated check items of the given CheckControl. Indices range from 1 to the number of check items. Illegal indices are ignored.
*/

selectRadioControlItem :: !Id ![Index] ![IOSt .1] -> IOSt .1
/* selectRadioControlItem marks the indicated radio item of a RadioControl, causing the mark of the previously marked radio item to disappear. The item is given by the Id of the RadioControl and its index position (counted from 1).
*/

selectPopUpControlItem :: !Id ![Index] ![IOSt .1] -> IOSt .1
/* selectPopUpControlItem marks the indicated popup item of a PopUpControl, causing the mark of the previously marked popup item to disappear. The item is given by the Id of the PopUpControl and its index position (counted from 1).
*/

moveControlViewFrame :: :Id Vector2 ![IOSt .1] -> IOSt .1
/* moveControlViewFrame moves the orientation of the CompoundControl over the given vector, and updates the control if necessary. The control frame is not moved outside the ViewDomain of the control. MoveControlViewFrame has no effect if the indicated control has no ControlDomain attribute.
*/

setControlViewDomain :: :Id ViewDomain ![IOSt .1] -> IOSt .1
/* setControlViewDomain sets the view domain of the indicated CompoundControl as
given. The control view frame is moved such that a maximum portion of the view
domain is visible. The control is not resized.
In case of unknown Ids, or non CompoundControls, setControlViewDomain has no
effect.

```haskell
setControlScrollFunction :: !Id Direction ScrollFunction !(IOSt .1) -> IOSt .1
/* setControlScrollFunction set the ScrollFunction of the indicated CompoundControl
in the given Direction if it has one.
In all other cases, setControlScrollFunction has no effect.
*/
```

```haskell
setControlTexts :: ![Id,String] !(IOSt .1) -> IOSt .1
setControlText :: !Id !String !(IOSt .1) -> IOSt .1
/* setControlText(s) sets the text of the indicated Text/Edit/ButtonControl(s).
If the indicated control is a Text/ButtonControl, then AltKey are interpreted
by the system.
If the indicated control is an EditControl, then the text is taken as it is.
*/
```

```haskell
setEditControlCursor :: !Id !Int !(IOSt .1) -> IOSt .1
/* setEditControlCursor sets the cursor at position @2 of the current content of
the EditControl.
In case @2<0, then the cursor is set at the start of the current content.
In case @2>size content, then the cursor is set at the end of the current
content.
*/
```

```haskell
setControlLooks :: ![Id,Bool,(Bool,Look)] !(IOSt .1) -> IOSt .1
setControlLook :: !Id !Bool (Bool,Look) !(IOSt .1) -> IOSt .1
/* setControlLook(s) sets the render/look attribute of the indicated
CompoundControl. If this concerns a transparant CompoundControl then it becomes non-transparant.
An indicated control is only redrawn if the first Boolean is True.
*/
```

```haskell
setSliderStates :: ![Id,IdFun SliderState] !(IOSt .1) -> IOSt .1
setSliderState :: !Id (IdFun SliderState) !(IOSt .1) -> IOSt .1
setSliderThumbs :: ![Id,Int] !(IOSt .1) -> IOSt .1
setSliderThumb :: !Id Int !(IOSt .1) -> IOSt .1
/* setSliderState(s) applies the function to the current SliderState of the indicated
SliderControl(s) and redraws the settings if necessary.
setSliderThumb(s) sets the new thumb value of the indicated SliderControl(s) and redraws the
settings if necessary.
*/
```

```haskell
appControlPicture :: !Id !(IdFun *Picture) !(IOSt .1) -> IOSt .1
accControlPicture :: !Id !((St *Picture .x)) !(IOSt .1)
               -> (!Maybe .x,!(IOSt .1))
/* (app/acc)ControlPicture applies the given drawing function to the Picture of
the indicated Custom/Button/CompoundControl. If the CompoundControl is
transparant, or the indicated control could not be found then this operation
has no effect. In that case, accControlPicture also returns Nothing.
*/
```

```haskell
updateControl :: !Id !(Maybe ViewFrame) !(IOSt .1) -> IOSt .1
/* updateControl applies the Look attribute function of the indicated
Compound/Custom/Button/CompoundControl. The Look attribute function is applied to the following arguments:
The current SelectState of the control, and
the UpdateState argument:
{oldFrame=viewframe,newFrame=viewframe,updArea=frame}
where viewframe is the current ViewFrame of the control;
and frame depends on the optional ViewFrame argument:
```
in case of (Just rectangle):
the intersection of viewframe and rectangle.
in case of Nothing:
viewframe.
updateControl has no effect in case of unknown controls, or if the indicated
control is not a (Compound/Custom(Button))Control, or the optional viewframe
argument is an empty rectangle.

/*

/* Access functions on WState. To read the state of a control, a WState is
required which can be obtained by the getWindow function. The WState value
represents the state of a window or dialogue at that particular moment.

/*
WState

getWindow :: !Id !(IOSt .!) \rightarrow (Maybe WState, !(IOSt .!))
getParentWindow :: !Id !(IOSt .!) \rightarrow (Maybe WState, !(IOSt .!))
/* getWindow returns a read-only WState for the indicated window.
In case the indicated window does not exist Nothing is returned.
getParentWindow returns a read-only WState for the parent window/dialogue
of the indicated control. In case the Id does not correspond with a
control, Nothing is returned.

/*
getControlTypes :: !WState \rightarrow [(ControlType,Maybe Id)]
getCompoundTypes :: !Id !WState \rightarrow [(ControlType,Maybe Id)]
/* getControlTypes
yields the list of ControlTypes of the component controls of this window.
getCompoundTypes
yields the list of ControlTypes of the component controls of this
CompoundControl.
For both functions (Just id) is yielded if the component control has a
(ControlId id) attribute, and Nothing otherwise. Component controls are not
collected recursively through CompoundControls.
If the indicated CompoundControl is not a CompoundControl, then [] is yielded.

/*
/* Functions that return the current state of controls.
For each access there is one singular and one plural version. In case of the
plural version the result list is of equal length as the argument Id list. Each
result list element corresponds in order with the argument Id list.
In both versions the first Boolean result is False in case of invalid Ids (if so
dummy values are returned - see comment).
Important: controls with no ControlId attribute, or illegal ids, can not be
found in the WState!

/*
getControlLayouts :: !Id !WState \rightarrow [(Bool,(Maybe ItemsPos,Vector2))]
getControlLayout :: !Id !WState \rightarrow (Bool,(Maybe ItemsPos,Vector2))
/* getControlLayout(s) yields (Just ControlPos) if the indicated control had a
ControlPos attribute and Nothing otherwise. The Vector2 offset is the exact
current location of the indicated control (LeftTop,OffsetVector offset).

/*
getControlViewSizes :: !Id !WState \rightarrow [(Bool,Size)]
getControlViewSize :: !Id !WState \rightarrow (Bool,Size)
getControlOuterSizes :: !Id !WState \rightarrow [(Bool,Size)]
getControlOuterSize :: !Id !WState \rightarrow (Bool,Size)
/* getControlViewSize(s) yields the current ViewFrame size of the indicated
control. Note that this is the exact size of the control for any control
other than the CompoundControl. In case of unknown Ids zero is returned.
getControlOuterSize(s) yields the current ControlOuterSize of the indicated
control. Note that this is the exact size of the control. In case of unknown
Ids zero is returned.
APPENDIX A. I/O LIBRARY

getControlSelectStates :: [Id] !WState \rightarrow [(Bool, SelectState)]
getControlSelectState :: !Id !WState \rightarrow (Bool, SelectState)
/* getControlSelectState(/s/) yields the current SelectState of the indicated control. In case of unknown Ids Able is returned. */

getControlShowStates :: [Id] !WState \rightarrow [(Bool, ShowState)]
getControlShowState :: !Id !WState \rightarrow (Bool, ShowState)
/* getControlShowState(/s/) yields True if the indicated control is visible, and False otherwise. The latter is also returned in case of unknown Ids. */

getControlTexts :: [Id] !WState \rightarrow [(Bool, Maybe String)]
getControlText :: !Id !WState \rightarrow (Bool, Maybe String)
/* getControlText(/s/) yields (Just text/) of the indicated (PopUp/Text/Edit/Button) Control. If the control is not such a control, then Nothing is yielded. */

getControlNrLines :: [Id] !WState \rightarrow [(Bool, Maybe NrLines)]
getControlNrLine :: !Id !WState \rightarrow (Bool, Maybe NrLines)
/* getControlNrLine(/s/) yields (Just nrlines/) of the indicated EditControl. If the control is not such a control, then Nothing is yielded. */

getControlLooks :: [Id] !WState \rightarrow [(Bool, Maybe (Bool, Look))] 
getControlLook :: !Id !WState \rightarrow (Bool, Maybe (Bool, Look))
/* getControlLook(/s/) yields the (render/look/) of the indicated (Custom/CustomButton/Compound)Control. If the control is not such a control, or is a transparant CompoundControl, then Nothing is yielded. */

getControlMinimumSizes :: [Id] !WState \rightarrow [(Bool, Maybe Size)]
getControlMinimumSize :: !Id !WState \rightarrow (Bool, Maybe Size)
/* getControlMinimumSize(/s/) yields (Just minimumsize/) if the indicated control had a ControlMinimumSize attribute and Nothing otherwise. */

getControlResizes :: [Id] !WState \rightarrow [(Bool, Maybe ControlResizeFunction)]
getControlResize :: !Id !WState \rightarrow (Bool, Maybe ControlResizeFunction)
/* getControlResize(/s/) yields (Just resizefunction/) if the indicated control had a ControlResize attribute and Nothing otherwise. */

getRadioControlItems :: [Id] !WState \rightarrow [(Bool, Maybe [String])] 
getRadioControlItem :: !Id !WState \rightarrow (Bool, Maybe [String])
/* getRadioControlItem(/s/) yields the TextLines of the items of the indicated RadioControl. If the control is not such a control, then Nothing is yielded. */

getRadioControlSelections :: [Id] !WState \rightarrow [(Bool, Maybe Index)]
getRadioControlSelection :: !Id !WState \rightarrow (Bool, Maybe Index)
/* getRadioControlSelection(/s/) yields the index of the selected radio item of the indicated RadioControl. If the control is not such a control, then Nothing is yielded. */

getCheckControlItems :: [Id] !WState \rightarrow [(Bool, Maybe [String])] 
getCheckControlItem :: !Id !WState \rightarrow (Bool, Maybe [String])
/* getCheckControlItem(/s/) yields the TextLines of the items of the indicated CheckControl. If the control is not such a control, then Nothing is yielded. */

getCheckControlSelections :: [Id] !WState \rightarrow [(Bool, Maybe [Index])] 
getCheckControlSelection :: !Id !WState \rightarrow (Bool, Maybe [Index])
/* getCheckControlSelection(/s/) yields the indices of the selected checkitems of the...
indicated CheckControl. If the control is not such a control, then Nothing is yielded.

getPopUpControlItems :: ![Id] !WState -> [(Bool,Maybe [String])]  
getPopUpControlItem :: ![Id] !WState -> (Bool,Maybe String)  
/* getPopUpControlItem(s) yields the TextLines of the items of the indicated PopUpControl. If the control is not such a control, then Nothing is yielded. */

getPopUpControlSelections :: ![Id] !WState -> [(Bool,Maybe Index)]  
getPopUpControlSelection :: ![Id] !WState -> (Bool,Maybe Index)  
/* getPopUpControlSelection(s) yields the Index of the indicated PopUpControl. If the control is not such a control, then Nothing is yielded. */

getSliderDirections :: ![Id] !WState -> [(Bool,Maybe Direction)]  
getSliderDirection :: ![Id] !WState -> (Bool,Maybe Direction)  
/* getSliderDirection(s) yields (Just Direction) of the indicated SliderControl. If the control is not such a control, then Nothing is yielded. */

getSliderStates :: ![Id] !WState -> [(Bool,Maybe SliderState)]  
getSliderState :: ![Id] !WState -> (Bool,Maybe SliderState)  
/* getSliderState(s) yields (Just SliderState) of the indicated SliderControl. If the control is not such a control, then Nothing is yielded. */

getControlViewFrames :: ![Id] !WState -> [(Bool,Maybe ViewFrame)]  
getControlViewFrame :: ![Id] !WState -> (Bool,Maybe ViewFrame)  
/* getControlViewFrame(s) yields (Just ViewFrame) of the indicated CompoundControl. If the control is not such a control, then Nothing is yielded. */

getControlItemSpaces :: ![Id] !WState -> [(Bool,Maybe (Int,Int))]  
getControlItemSpace :: ![Id] !WState -> (Bool,Maybe (Int,Int))  
/* getControlItemSpace(s) yields (Just (horizontal space,vertical space)) of the indicated (Compound/Layout)Control. If the control is not such a control, then Nothing is yielded. */

getControlMargins :: ![Id] !WState -> [(Bool,Maybe (Int,Int))]  
getControlMargin :: ![Id] !WState -> (Bool,Maybe (Int,Int))  
/* getControlMargins yields (Just (ControlHMargin,ControlVMargin)) of the indicated (Compound/Layout)Control. If the control is not such a control, then Nothing is yielded. */
A.5 StdControlAttribute

definition module StdControlAttribute

// Clean Standard Object I/O library, version 1.2
// StdControlAttribute specifies which ControlAttributes are valid for each of the standard controls.
// Basic comparison operations and retrieval functions are also included.
// 
import StdControlDef

/* The following functions specify the valid attributes for each standard control.
*/
isValidButtonControlAttribute :: !(ControlAttribute . st) -> Bool
/* ButtonControl (y = valid, . = invalid)
  ControlActivate . | ControlKeyboard . | ControlPos y |
  ControlFunction y | ControlMinimumSize . | ControlSelectState y |
  ControlHide y | ControlModFunction y | ControlTip y |
  ControlHMargin . | ControlMouse . | ControlViewDomain . |
  ControlHScroll . | ControlOrigin . | ControlViewSize . |
  ControlId y | ControlouterSize . | ControlVMargin . |
  ControlWidth y |
*/
isValidCheckControlAttribute :: !(ControlAttribute . st) -> Bool
/* CheckControl (y = valid, . = invalid)
  ControlActivate . | ControlKeyboard . | ControlPos y |
  ControlFunction y | ControlMinimumSize . | ControlSelectState y |
  ControlHide y | ControlModFunction y | ControlTip y |
  ControlHMargin . | ControlMouse . | ControlViewDomain . |
  ControlHScroll . | ControlOrigin . | ControlViewSize . |
  ControlId y | ControlouterSize . | ControlVMargin . |
  ControlWidth . |
*/
isValidCompoundControlAttribute :: !(ControlAttribute . st) -> Bool
/* CompoundControl (y = valid, . = invalid)
  ControlActivate y | ControlKeyboard y | ControlPos y |
  ControlDeactivate y | ControlLook y | ControlResize y |
  ControlFunction y | ControlMinimumSize y | ControlSelectState y |
  ControlHide y | ControlModFunction y | ControlTip y |
  ControlHMargin y | ControlMouse y | ControlViewDomain y |
  ControlHScroll y | ControlOrigin y | ControlViewSize y |
  ControlId y | ControlouterSize y | ControlVMargin y |
  ControlItemsSpace y | ControlPen y | ControlVScroll y |
  ControlWidth . |
*/
isValidCustomButtonControlAttribute :: !(ControlAttribute . st) -> Bool
/* CustomButtonControl (y = valid, . = invalid)
  ControlActivate . | ControlKeyboard . | ControlPos y |
  ControlDeactivate . | ControlLook . | ControlResize y |
  ControlFunction y | ControlMinimumSize y | ControlSelectState y |
  ControlHide y | ControlModFunction y | ControlTip y |
  ControlHMargin . | ControlMouse . | ControlViewDomain . |
  ControlHScroll . | ControlOrigin . | ControlViewSize . |
A.5. **STDCONTROLATTRIBUTE**

```plaintext
isValidCustomControlAttribute :: ! (ControlAttribute . st) -> Bool
isValidEditControlAttribute :: ! (ControlAttribute . st) -> Bool
isValidLayoutControlAttribute :: ! (ControlAttribute . st) -> Bool
isValidPopUpControlAttribute :: ! (ControlAttribute . st) -> Bool
isValidRadioControlAttribute :: ! (ControlAttribute . st) -> Bool
```
isValidSliderControlAttribute :: ! (ControlAttribute . st) -> Bool
/* SliderControl */
ControlDeactivate . | ControlLook . | ControlResize y . |
ControlFunction . | ControlMinimumSize . | ControlSelectState y . |
ControlHMargin . | ControlMouse . | ControlViewDomain . |
ControlHScroll . | ControlOrigin . | ControlViewSize . |
ControlId y . | ControlOuterSize . | ControlVMargin . |

isValidTextControlAttribute :: ! (ControlAttribute . st) -> Bool
/* TextControl */
ControlFunction . | ControlMinimumSize . | ControlSelectState . |
ControlHMargin . | ControlMouse . | ControlViewDomain . |
ControlHScroll . | ControlOrigin . | ControlViewSize . |
ControlId y . | ControlOuterSize . | ControlVMargin . |

/* The following functions return True only iff the attribute equals the
indicated name. */
isControlActivate :: ! (ControlAttribute . st) -> Bool
isControlDeactivate :: ! (ControlAttribute . st) -> Bool
isControlFunction :: ! (ControlAttribute . st) -> Bool
isControlHide :: ! (ControlAttribute . st) -> Bool
isControlHMargin :: ! (ControlAttribute . st) -> Bool
isControlHScroll :: ! (ControlAttribute . st) -> Bool
isControlId :: ! (ControlAttribute . st) -> Bool
isControlItemSpace :: ! (ControlAttribute . st) -> Bool
isControlKeyboard :: ! (ControlAttribute . st) -> Bool
isControlLook :: ! (ControlAttribute . st) -> Bool
isControlMinimumSize :: ! (ControlAttribute . st) -> Bool
isControlModsFunction :: ! (ControlAttribute . st) -> Bool
isControlMouse :: ! (ControlAttribute . st) -> Bool
isControlOrigin :: ! (ControlAttribute . st) -> Bool
isControlOuterSize :: ! (ControlAttribute . st) -> Bool
isControlPen :: ! (ControlAttribute . st) -> Bool
isControlPos :: ! (ControlAttribute . st) -> Bool
isControlResize :: ! (ControlAttribute . st) -> Bool
isControlSelectState :: ! (ControlAttribute . st) -> Bool
isControlTip :: ! (ControlAttribute . st) -> Bool
isControlViewDomain :: ! (ControlAttribute . st) -> Bool
isControlVMargin :: ! (ControlAttribute . st) -> Bool
isControlVScroll :: ! (ControlAttribute . st) -> Bool
isControlWidth :: ! (ControlAttribute . st) -> Bool

/* The following functions return the attribute value if appropriate.
THESE ARE PARTIAL FUNCTIONS! They are only defined on the corresponding
attribute. */
getControlActivateFun :: ! (ControlAttribute . st) -> IdFun . st
getControlDeactivateFun :: ! (ControlAttribute . st) -> IdFun . st
getControlFun :: ! (ControlAttribute . st) -> IdFun . st
A.5. **STDCONTR OLA TTRIBUTE**

```
getControlHMarginAtt :: (ControlAttribute .st) -> (Int,Int)
getControlHScrollFun :: (ControlAttribute .st) -> ScrollFunction
getControlIdAtt :: (ControlAttribute .st) -> Id
getControlItemSpaceAtt :: (ControlAttribute .st) -> (Int,Int)
getControlKeyboardAtt :: (ControlAttribute .st) -> (KeyboardStateFilter, SelectState, KeyboardFunction .st)
getControlLookAtt :: (ControlAttribute .st) -> (Bool,Look)
getControlMinimumSizeAtt :: (ControlAttribute .st) -> Size
getControlModesFun :: (ControlAttribute .st) -> ModifierFunction .st
getControlMouseAtt :: (ControlAttribute .st) -> (MouseStateFilter, SelectState, MouseFunction .st)
getControlOriginAtt :: (ControlAttribute .st) -> Point2
getControlOuterSizeAtt :: (ControlAttribute .st) -> Size
getControlPenAtt :: (ControlAttribute .st) -> [PenAttribute]
getControlPosAtt :: (ControlAttribute .st) -> ItemPos
getControlResizeFun :: (ControlAttribute .st) -> ControlResizeFunction
getControlSelectStateAtt :: (ControlAttribute .st) -> SelectState
getControlTipAtt :: (ControlAttribute .st) -> String
getControlViewDomainAtt :: (ControlAttribute .st) -> ViewDomain
getControlViewSizeAtt :: (ControlAttribute .st) -> Size
getControlVMarginAtt :: (ControlAttribute .st) -> (Int,Int)
getControlVScrollFun :: (ControlAttribute .st) -> ScrollFunction
getControlWidthAtt :: (ControlAttribute .st) -> ControlWidth
```
### A.6 StdControlClass

**definition module StdControlClass**

```plaintext
// ******************************************************************************
// Clean Standard Object I/O library, version 1.2
// StdControlClass define the standard set of controls instances.
// ******************************************************************************

import StdControlDef
from windowhandle import ControlState
from StdPSt import PSt, IOST

class Controls cdef where
    controlToHandles :: ![](cdef .ls (PSt .l)) !(PSt .l)
                        -> (![](ControlState .ls (PSt .l]), !PSt .l))
    getControlType :: (cdef .ls .pst)
                        -> ControlType

instance Controls (AddLS c) | Controls c
instance Controls (NewLS c) | Controls c
instance Controls (ListLS c) | Controls c
instance Controls NilLS
instance Controls ((:+:) c1 c2) | Controls c1 & Controls c2

instance Controls ButtonControl
instance Controls CheckControl
instance Controls (CompoundControl c) | Controls c
instance Controls CustomButtonControl
instance Controls CustomControl
instance Controls EditControl
instance Controls (LayoutControl c) | Controls c
instance Controls PopUpControl
instance Controls RadioControl
instance Controls SliderControl
instance Controls TextControl
```
A.7 StdControlDef

definition module StdControlDef

// ******************************************************************************
// Clean Standard Object I/O library, version 1.2
//
// StdControl contains the types to define the standard set of controls.
// ******************************************************************************

import StdIOCommon, StdPictureDef

:: ButtonControl ls pst
  = ButtonControl String  [ControlAttribute *(ls.pst)]
:: CheckControl ls pst
  = CheckControl  [CheckControlItem *(ls.pst)] RowsOrColumns
:: CompoundControl c ls pst
  = CompoundControl *(c ls pst) [ControlAttribute *(ls.pst)]
:: CustomButtonControl ls pst
  = CustomButtonControl Size Look [ControlAttribute *(ls.pst)]
:: CustomControl ls pst
  = CustomControl Size Look [ControlAttribute *(ls.pst)]
:: EditControl ls pst
  = EditControl String ControlWidth NrLines [ControlAttribute *(ls.pst)]
:: LayoutControl c ls pst
  = LayoutControl *(c ls pst) [ControlAttribute *(ls.pst)]
:: PopUpControl ls pst
  = PopUpControl  [PopUpControlItem *(ls.pst)] Index
:: RadioControl ls pst
  = RadioControl [RadioControlItem *(ls.pst)] RowsOrColumns Index
:: SliderControl ls pst
  = SliderControl Direction ControlWidth SliderState [SliderAction *(ls.pst)]
:: TextControl ls pst
  = TextControl String [ControlAttribute *(ls.pst)]
:: CheckControlItem st := (String, Maybe ControlWidth, MarkState, IdFun st)
:: PopUpControlItem st := (String, IdFun st)
:: RadioControlItem st := (String, Maybe ControlWidth, IdFun st)
:: NrLines := Int
:: RowsOrColumns
  = Rows Int
    | Columns Int
:: ControlWidth  // The width of the control:
  = PixelWidth Int  // the exact number of pixels
  | TextWidth String  // the exact string width in dialog font
  | ContentWidth String  // width of the control as if string is its content
:: ControlAttribute st  // Default:
  // General control attributes:
  = ControlActivate  // id
  | ControlDeactivate  // id
  | ControlFunction  // id
  | ControlHide  // initially visible
  | ControlId  // no id
  | ControlKeyboard KeyboardStateFilter SelectState (KeyboardFunction st)
  = ControlMinimumSize Size  // no keyboard input/overruled
  | ControlModifiersFunction (ModifiersFunction st)  // no mouse input/overruled
  | ControlMouse MouseStateFilter SelectState (MouseFunction st)
| ControlPen    | PenAttribute              | // default pen attributes |
| ControlPos    | ItemPos                   | // (right to previous, zero) |
| ControlResize | ControlResizeFunction     | // no resize |
| ControlSelectState | SelectState               | // control able |
| ControlTip    | String                    | // no tip |
| ControlWidth  | ControlWidth              | // system derived |

// For CompoundControls only:
| ControlHMargin | Int Int                   | // system dependent |
| ControlHScroll | ScrollFunction            | // no horizontal scrolling |
| ControlItemSpace | Int Int                 | // system dependent |
| ControlLook    | Bool Look                 | // control is transparent |
| ControlOrigin  | Point2                    | // left top of ViewDomain |
| ControlOuterSize | Size                   | // enclose elements |
| ControlViewDomain | ViewDomain             | // (zero, max range) |
| ControlViewSize | Size                     | // enclose elements |
| ControlVMargin | Int Int                   | // system dependent |
| ControlVScroll | ScrollFunction            | // no vertical scrolling |

: ControlResizeFunction
  ::= Size -> Size -> // current control outer size
  | Size -> Size     // old parent view size
  | Size -> Size     // new parent view size
  | Size             // new control outer size

: ControlType
  ::= String
A.8 StdControlReceiver

definition module StdControlReceiver

//******************************************************************************
// Clean Standard I/O library, version 1.2
//
// StdControlReceiver defines Receiver(2) controls instances.
//******************************************************************************

import StdReceiverDef, StdControlClass

instance Controls (Receiver m )
instant Controls (Receiver2 m r)
A.9 StdEventTCP

definition module StdEventTCP

// **********************************************************************************************************
// Clean Standard Object I/O library, version 1.2
// StdEventTCP provides functions for using event driven TCP
// **********************************************************************************************************

import StdChannels, StdTCPDef
from StdString import String
from StdReceiver import Receivers, ReceiverType
from StdPSt import PSt, IOSt
from StdIOCommon import ErrorReport
from tcp_bytestreams import TCP_SCharStream_
from StdPStClass import FileEnv, Files, TimeEnv, Date, Tick, Time

instance ChannelEnv (PSt .l/), (IOSt .l/)
instance Receivers TCP_ListenerReceiver
instance Receivers TCP_Receiver
instance Receivers TCP_CharReceiver

openSendNotifier :: .ls !(SendNotifier (*ch .a) .ls (PSt .l))
                 !((PSt .l))
                 -> (!ErrorReport,!*(*ch .a),!PSt .l)
                 | accSChannel ch & Send ch
/* opens a send notifier, which informs the application, that sending on the channel is again possible due to flow conditions. Possible error reports are NoError and ErrorNotifierOpen */

closeSendNotifier :: !(*ch .a),!IOSt .l)
                    -> (!(*ch .a), !IOSt .l)
                    | accSChannel ch
/* to close a send notifier. This function will be called implicitly if a send channel is closed, so there is no need to do it explicitly then. */

lookupIPAddress_async :: !String !(InetLookupFunction (PSt .l)) !(PSt .l)
                        -> (PSt .l)
/* lookupIPAddress_async asynchronously looks up an IP address. The String can be in dotted decimal form or alphanumerical. The InetLookupFunction will be called with the IP address, if this address was found, otherwise with Nothing. */

connectTCP_async :: !(!IPAddress,!Port) !(InetConnectFunction (PSt .l))
                    !!(PSt .l)
                    -> (PSt .l)
/* connectTCP_async asynchronously tries to establish a new connection. The InetConnectFunction will be called with the new duplex channel if this attempt was successful, otherwise with Nothing */

class accSChannel ch :: (TCP_SChannel -> (x, TCP_SChannel)) *!(*ch .a)
                        -> (x, *!(*ch .a))
/* This overloaded function supports the openSendNotifier function. It applies an access function on the underlying TCP_SChannel */

instance accSChannel TCP_SChannel_
instance accSChannel TCP_SCharStream_
A.10  StdFileSelect

definition module StdFileSelect

// Clean Standard Object I/O library, version 1.2
// StdFileSelect defines the standard file selector dialogue.
// ******************************************************************

import StdMaybe, StdString

class FileSelectEnv env where
    selectInputFile :: env -> (!Maybe String, env)
    selectOutputFile :: !String !String env -> (!Maybe String, env)
    selectDirectory :: env -> (!Maybe String, env)

/* selectInputFile
   opens a dialogue in which the user can browse the file system to select an
   existing file.
   If a file has been selected, the String result contains the complete
   pathname of the selected file.
   If the user has not selected a file, Nothing is returned.
selectOutputFile
   opens a dialogue in which the user can browse the file system to save a
   file.
   The first argument is the prompt of the dialogue (default: "Save As:")
   The second argument is the suggested filename.
   If the indicated directory already contains a file with the indicated name,
   selectOutputFile opens a new dialogue to confirm overwriting of the existing
   file.
   If either this dialogue is not confirmed or browsing is cancelled then
   Nothing is returned, otherwise the String result is the complete pathname of
   the selected file.
selectDirectory
   opens a dialogue in which the user can browse the file system to select a
   directory.
   If a directory has been selected, the String result contains the complete
   pathname of the selected directory.
   If the user has not selected a directory, Nothing is returned. */

instance FileSelectEnv World
A.11 StdId

definition module StdId

// Clean Standard Object I/O library, version 1.2
// StdId specifies the generation functions for identification values.

from StdMaybe import Maybe, Just, Nothing
from id import Id, RId, R2Id, RIdtoId, R2IdtoId, toString, ==
from iostate import PSt, IOSt

class Ids env where
  openId :: !env -> (!Id, !env)
  openIds :: !Int !env -> (![Id], !env)
  openRId :: !env -> (!RId m, !env)
  openRIds :: !Int !env -> (![RId m], !env)
  openR2Id :: !env -> (!R2Id m r, !env)
  openR2Ids :: !Int !env -> (![R2Id m r], !env)

/* There are three types of identification values:
   - RId m: for uni-directional message passing (see StdReceiver)
   - R2Id m r: for bi-directional message passing (see StdReceiver)
   - Id: for all other Object I/O library components
     If each generation function there are two variants:
     - to create exactly one identification value.
     - to create a number of identification values.
     If the integer argument <=0, then an empty list of identification values
     is generated.
*/

instance Ids World,
  IOSt .l, PSt .l

getParentId :: !Id !(IOSt .l) -> (!Maybe Id, !IOSt .l)
/* getParentId returns the Id of the parent top-level GUI object
   of the GUI component identified by the argument Id.
   If the GUI component could not be found then Nothing is returned.
*/
A.12 StdIO

definition module StdIO

// ******************************************************************************************************************************************
// Clean Standard Object I/O library, version 1.2
// StdIO contains all definition modules of the Object I/O library.
// ******************************************************************************************************************************************

import

StdId, // The operations that generate identification values
StdIOMixin, // Function and type definitions used in the library
StdIOCommon, // Function and type definitions used in the library
StdKey, // Function and type definitions on keyboard
StdMaybe, // The Maybe data type
StdPSt, // Operations on PSt that are not device related
StdPStClass, // PSt/ISt instances of common classes
StdSystem, // System dependent operations

StdFileSelect, // File selector dialogues

StdPictureDef, // Type definitions for picture handling
StdPicture, // Picture handling operations
StdBitmap, // Defines an instance for drawing bitmaps

StdProcessDef, // Type definitions for process handling
StdProcessAttribute, // ProcessAttribute access operations
StdProcess, // Process handling operations

StdClipboard, // Clipboard handling operations

StdPrint, // General printing functions
StdPrintText, // Specialised text printing functions

StdControlDef, // Type definitions for controls
StdControlAttribute, // ControlAttribute access operations
StdControlClass, // Standard controls class instances
StdControlReceiver, // Receiver controls class instances
StdControl, // Control handling operations

StdMenuDef, // Type definitions for menus
StdMenuAttribute, // MenuAttribute access operations
StdMenuElementClass, // Standard menus class instances
StdMenuReceiver, // Receiver menus class instances
StdMenuElement, // Menu element handling operations
StdMenu, // Menu handling operations

StdReceiverDef, // Type definitions for receivers
StdReceiverAttribute, // ReceiverAttribute access operations
StdReceiver, // Receiver handling operations

StdTimerDef, // Type definitions for timers
StdTimerAttribute, // TimerAttribute access operations
StdTimerElementClass, // Standard timer class instances
StdTimerReceiver, // Receiver timer class instances
StdTimer, // Timer handling operations
StdTime, // Time related operations

StdWindowDef, // Type definitions for windows
StdWindowAttribute, // WindowAttribute access operations
StdWindow // Window handling operations
A.13 StdIOBasic

definition module StdIOBasic

// ********************************************************************************
// Clean Standard Object I/O library, version 1.2
// StdIOBasic defines basic types and access functions for the I/O library.
// ********************************************************************************

import StdOverloaded, StdString

/* General type constructors for composing context-independent data structures. */
:: :: t1 t2 = (::) infixr 9 t1 t2

/* General type constructors for composing context-dependent data structures. */
:: :: t1 t2 cs = (::) infixr 9 (t1 cs) (t2 cs)
:: ListCS t cs = ListCS [t cs]
:: NilCS cs = NilCS

/* General type constructors for composing local and context-dependent data structures. */
:: :: t1 t2 ls cs = (:+) infixr 9 (t1 ls cs) (t2 ls cs)
:: ListLS t ls cs = ListLS [t ls cs]
:: NilLS ls cs = NilLS
:: NewLS t ls cs = E.nev: {newLS::nev, newDef:: t nev cs}
:: AddLS t ls cs = E.add: {addLS::add, addDef:: t *(add,ls) cs}
noLS :: (a -> b) (c , a) -> (c , b) // Lift function a -> b
     to (c ,a) -> (c ,b)
noLS1: (x -> a -> b) .x (c , a) -> (c , b) // Lift function x -> a -> b
     to x -> (c ,a) -> (c ,b)

:: Index := Int
:: Title := String
:: Vector2 = {vx::!Int,vy::!Int}

instance = Vector2 // @1@2 = zero
instance + Vector2 // {vx=@1.vx+@2.vx,vy=@1.vy+@2.vy}
instance - Vector2 // {vx=@1.vx-@2.vx,vy=@1.vy-@2.vy}
instance zero Vector2 // {vx=0,vy=0}
instance toVector Vector2 // zero@1
instance toString Vector2

class toVector x :: !x -> Vector2

:: Size = {w ::!Int,h ::!Int}

instance = Size // @1.w=@2.w h@1.h=@2.h
instance zero Size // {w=0,h=0}
instance toVector Size // {w,h}->{w,v,vy=0}
instance toString Size

:: Point2
- { x :: !Int
  , y :: !Int
}
:: Rectangle
- { corner1 :: !Point2
  , corner2 :: !Point2
}

instance == Point2 // @1-@2==zero
instance + Point2 // {x=@1.x+@2.x,y=@1.y+@2.y}
instance - Point2 // {x=@1.x-@2.x,y=@1.y-@2.y}
instance zero Point2 // {x=0,y=0}
instance toVector Point2 // {x,y}->(vx,vx-vy)
instance toString Point2

instance == Rectangle // @1.corner1==@2.corner1
instance zero Rectangle // {corner1==zero,corner2==zero}
instance toString Rectangle

rectangleSize :: !Rectangle -> Size // {w=abs(@1.corner1@2.corner2).x,
// h=abs(@1.corner1@1.corner2).y}
movePoint :: !Vector2 !Point2 -> .Point2 // {vx,vy} {x,y} -> {vx+x,vy+y}

:: IdFun st :: st -> st
:: Void = Void
A.14 StdIOCommon

definition module StdIOCommon

// Clean Standard Object I/O library, version 1.2
// StdIOCommon defines common types and access functions for the I/O library.

import StdOverloaded, StdString
import StdBitmap, StdIOBasic, StdKey, StdMaybe
from id import Id, RId, R2Id, RIdtoId, R2IdtoId, toString, =

/* The SelectState and MarkState types. */
:: SelectState - Able | Unable
:: MarkState - Mark | NoMark

enabled :: !SelectState -> Bool // @1 == Able
marked :: !MarkState -> Bool // @1 == Mark

instance = SelectState // Constructor equality
instance = MarkState // Constructor equality
instance ~ SelectState // Able <-> Unable
instance ~ MarkState // Mark <-> NoMark
instance toString SelectState
instance toString MarkState

/* The KeyboardState type. */
:: KeyboardState
  - CharKey Char KeyState // ASCII character input
  | SpecialKey SpecialKey KeyState Modifiers
    // Special key input
  | KeyLost
    // Key input lost while key was down
  :: KeyState
    - KeyDown IsRepeatKey // Key is down
    | KeyUp // Key goes up
  :: IsRepeatKey
    // Flag on key down:
    := Bool // True iff key is repeating
  :: Key
    - IsCharKey Char
    | IsSpecialKey SpecialKey
  :: KeyboardStateFilter // Predicate on KeyboardState:
    := KeyboardState -> Bool // evaluate KeyFunction only if True

getKeyboardStateKeyStat : !KeyboardState -> KeyState // KeyUp if KeyLost
getKeyboardStateKey : !KeyboardState -> Maybe Key // Nothing if KeyLost

instance = KeyboardState // Equality on KeyboardState
instance = KeyState // Equality on KeyState
instance toString KeyboardState
instance toString KeyState

/* The MouseState type. */
:: MouseState
  - MouseMove Point2 Modifiers // Mouse is up (position, modifiers)
  | MouseDown Point2 Modifiers Int // Mouse goes down (and nr down)
  | MouseDrag Point2 Modifiers // Mouse is down (position, modifiers)
:: MouseState
  = ButtonState
  = ButtonStillUp
  = ButtonDown
  = ButtonDoubleDown
  = ButtonTripleDown
  = ButtonStillDown
  = ButtonUp

:: MouseStateModifiers

:: MouseStateButtonState

:: MouseStateFilter
// Predicate on MouseState:
instance == MouseState // Equality on MouseState
instance == ButtonState // Constructor equality
instance toString MouseState
instance toString ButtonState

/* The SliderState type. */
:: SliderState
  = { sliderMin :: !Int
       , sliderMax :: !Int
       , sliderThumb :: !Int
     }

instance == SliderState // @1.sliderMin == @2.sliderMin
instance == SliderState // @1.sliderMax == @2.sliderMax
instance == SliderState // @1.sliderThumb == @2.sliderThumb

instance toString SliderState

/* The UpdateState type. */
:: UpdateState
  = { oldFrame :: !ViewFrame
       , newFrame :: !ViewFrame
       , updArea :: !UpdateArea
     }

instance == UpdateState // ViewDomain == Rectangle
instance == UpdateState // ViewFrame == Rectangle
instance == UpdateState // UpdateArea == [ViewFrame]

instance toString UpdateState
RectangleToUpdateState :: !Rectangle -> UpdateState
// r -> {oldFrame=newFrame=r, updArea=[r]}

/* viewDomainRange defines the minimum and maximum values for ViewDomains.
viewFrameRange defines the minimum and maximum values for ViewFrames. */
viewDomainRange ::= { corner1 = {x = 0-2·30,y = 0-2·30}
                      , corner2 = {x = 2·30,y = 2·30 }
                    }
viewFrameRange ::= { corner1 = {x = 1-2·31,y = 1-2·31}
                      , corner2 = {x = 2·31-1,y = 2·31-1}
                    }

/* Modifiers indicates the meta keys that have been pressed (True) or not (False). */
:: Modifiers
  = { shiftDown :: !Bool // True iff shift down
       , optionDown :: !Bool // True iff option down
     }
APPENDIX A. I/O LIBRARY

. commandDown :: !Bool // True iff command down
. controlDown :: !Bool // True iff control down
. altDown :: !Bool // True iff alt down

// Constants to check which of the Modifiers are down.

NoModifiers ::= {shiftDown = False
  , optionDown = False
  , commandDown= False
  , controlDown= False
  , altDown  = False
}
ShiftOnly ::= {shiftDown = True
  , optionDown = False
  , commandDown= False
  , controlDown= False
  , altDown  = False
}
OptionOnly ::= {shiftDown = False
  , optionDown = True
  , commandDown= False
  , controlDown= False
  , altDown  = True
}
CommandOnly ::= {shiftDown = False
  , optionDown = False
  , commandDown= True
  , controlDown= True
  , altDown  = False
}
ControlOnly ::= {shiftDown = False
  , optionDown = False
  , commandDown= False
  , controlDown= True
  , altDown  = True
}
AltOnly ::= {shiftDown = False
  , optionDown = True
  , commandDown= False
  , controlDown= False
  , altDown  = True
}

instance = Modifiers
instance toString Modifiers

// The layout language used for windows and controls. */
:: ItemPos
  ::= ( ItemLoc
  , ItemOffset
  )
:: ItemLoc
  // Absolute:
    = Fix
  // Relative to corner:
    | LeftTop
    | RightTop
    | LeftBottom
    | RightBottom
  // Relative in next line:
    | Left
    | Center
    | Right
  // Relative to other item:
    | LeftOf Id
```plaintext
// Relative to previous item:
| LeftToPrev
| RightToPrev
| AbovePrev
| BelowPrev

:: ItemOffset
   = NoOffset              // Shorthand for OffsetVector zero
   | OffsetVector Vector2   // A constant offset vector
   | OffsetFun ParentIndex OffsetFun // Offset depends on orientation

:: ParentIndex
   ::= Int                // The number of parents (1..)

:: OffsetFun
   ::= (ViewDomain,Point2) -> Vector2 // Current view domain and origin

instance zero ItemOffset // zero == NoOffset
instance == ItemLoc      // Constructor and value equality
instance toString ItemLoc // Constructor and value as String

/*/ The Direction type.  */
:: Direction
   = Horizontal
   | Vertical

instance == Direction     // Constructor equality
instance toString Direction // Constructor as String

/*/ The CursorShape type.  */
:: CursorShape
   = StandardCursor
   | BusyCursor
   | IBeamCursor
   | CrossCursor
   | FatCrossCursor
   | ArrowCursor
   | HiddenCursor

instance == CursorShape   // Constructor equality
instance toString CursorShape // Constructor as String

/*/ Document interface of interactive processes.  */
:: DocumentInterface
   = NDI              // No Document Interface
   | SDI              // Single Document Interface
   | MDI              // Multiple Document Interface

instance == DocumentInterface // Constructor equality
instance toString DocumentInterface // Constructor as String

/*/ Process attributes.  */
:: ProcessAttribute st               // Default:
   = ProcessActivate (IdFun st)       // No action on activate
   | ProcessDeactivate (IdFun st)      // No action on deactivate
   | ProcessClose (IdFun st)           // Process is closed

// Attributes for (M/S)DI process only:
| ProcessOpenFiles (ProcessOpenFilesFunction st) // Request to open files
```
| ProcessWindowPos ItemPos   // Platform dependent |
| ProcessWindowSize Size    // Platform dependent |
| ProcessWindowResize (ProcessWindowResizeFunction st) // Platform dependent |
| ProcessToolBar [ToolbarItem st] // Process has no toolbar |

// Attributes for MDI processes only:
| ProcessNoWindowMenu // Process has WindowMenu |

:: ProcessWindowResizeFunction st
:: := Size // Old ProcessWindow size
    -> Size // New ProcessWindow size
    -> st -> st
:: ProcessOpenFilesFunction st
:: := [String] // The filenames to open
    -> st -> st

:: ToolbarItem st
:: = ToolbarItem Bitmap (Maybe String) (IdFun st)
| ToolbarSeparator

/* Frequently used function types. */
:: ModifierFunction st := Modifiers -> st -> st
:: MouseFunction st := MouseState -> st -> st
:: KeyboardFunction st := KeyboardState -> st -> st
:: SliderAction st := SliderMove -> st -> st
:: SliderMove
    = SliderIncSmall
    | SliderDecSmall
    | SliderIncLarge
    | SliderDecLarge
    | SliderThumb Int

instance toString SliderMove

/* Scrolling function. */
:: ScrollFunction
:: := ViewFrame -> // Current view
    SliderState -> // Current state of scrollbar
    SliderMove -> // Action of the user
    Int // New thumb value

stdScrollFunction :: !Direction !Int -> ScrollFunction
/* stdScrollFunction direction d implements standard scrolling behaviour:
- direction indicates scrolling for Horizontal or Vertical scroll bar.
- d is the stepsise with which to scroll (taken absolute).
stdScrollFunction lets the system scroll as follows:
- Slider(Inc/Dec)Small: d
- Slider(Inc/Dec)Large: viewFrame size modulo d
- SliderThumb x: x modulo d */

/* Standard GUI object rendering function. */
:: Look
:: := SelectState -> // Current SelectState of GUI object
    UpdateState -> // The area to be rendered
    *Picture -> *Picture // The rendering action

stdUnfillNewFrameLook :: SelectState !UpdateState !Picture -> *Picture
stdUnfillUpdAreaLook :: SelectState !UpdateState !Picture -> *Picture
/* Two convenience functions for simple Look functions:
stdUnfillNewFrameLook _ (newFrame) = unfill newFrame */
stdUnfillUpdAreaLook _ {updArea} = seq (map unfill updArea)

/*

/* Common error report types. */

:: ErrorReport
   -- NoError // Usual cause: Everything went alright
   | ErrorViolateDI // Violation against DocumentInterface
   | ErrorIdsInUse // Object contains Ids that are bound
   | ErrorUnknownObject // Object can not be found
   | ErrorNotifierOpen // It was tried to open a second send notifier
   | OtherError !String // Other kind of error

instance == ErrorReport // Constructor equality
instance toString ErrorReport // Constructor as String

:: OkBool // True iff the operation was successful
   == Bool
A.15 StdKey

definition module StdKey

// **************************************************************
// Clean Standard Object I/O library, version 1.2
// 
// StdKey defines the special keys for the Object I/O library.
// **************************************************************

from key import SpecialKey, 
    BackSpaceKey, BeginKey, 
    ClearKey, 
    DeleteKey, DmDownKey, 
    EndKey, EnterKey, EscapeKey, 
    F1Key, F2Key, F3Key, F4Key, F5Key, F6Key, 
    F7Key, F8Key, F9Key, F10Key, 
    F11Key, F12Key, F13Key, F14Key, F15Key, 
    HelpKey, 
    LeftKey, 
    PgDownKey, PgUpKey, 
    RightKey, 
    UpKey, 
    --, toString
 StdMaybe

definition module StdMaybe

// Clean Standard Object I/O library, version 1.2
// StdMaybe defines the Maybe type.

import StdFunc
import StdOverloaded
import StdIOBasic
import IdFun

type Maybe a = Just a | Nothing

-- Just x
| Nothing

isJust :: (Maybe a) -> Bool
where
  isJust (Just _) = True
  isJust _ = False

isNothing :: (Maybe a) -> Bool
where
  isNothing _ = not (isJust _)

fromJust :: (Maybe a) -> a
where
  fromJust (Just x) = x

// for possibly unique elements:

isJust :: (Maybe a) -> Bool
where
  u_isJust (Just x) = (fst (f x), Just (snd (f x)))
  u_isNothing (Nothing) = (Nothing, Nothing)

accMaybe :: (a -> b) -> (Maybe a, Maybe b) -> (Maybe a, Maybe b)
where
  accMaybe f (Just x) = (Just (fst (f x)), Just (snd (f x)))
  accMaybe f Nothing = (Nothing, Nothing)

mapMaybe :: (a -> b) -> (Maybe a) -> Maybe b
where
  mapMaybe f (Just x) = Just (f x)
  mapMaybe f Nothing = Nothing

instance -- (Maybe a) | -- x
where
  Nothing <= a = a

// Just a <= Just b <= a-b
A.17 StdMenu

definition module StdMenu

// *****************************************
// Clean Standard Object I/O library, version 1.2
// StdMenu defines functions on menus.
// *****************************************

import StdMenuElementClass
from iostate import PSt, IOSt

// Operations on unknown Ids are ignored.

class Menus mdef where
openMenu :: Id !(mdef .ls (PSt .1)) !(PSt .1) -> ((ErrorReport, PSt .1))
getMenuType :: (mdef .ls .pst) -> MenuType

/*
Open the given menu definition for this interactive process.

openMenu may not be permitted to open a menu depending on its DocumentInterface
(see the comments at the shareProcesses instances in module StdProcess).
In case a menu with the same Id is already open then nothing happens. In case the
menu has the WindowMenuId Id then nothing happens. In case the menu does
not have an Id, it will obtain an Id which is fresh with respect to the
current set of menus. The Id can be reused after closing this menu. In case
menus elements are opened with duplicate Ids, the menu will not be opened.
In case the menu definition does not have a MenuIndex attribute (see StdMenuDef)
it will be opened behind the last menu. In case the menu definition has a
MenuIndex attribute it will be placed behind the menu indicated by the
integer index.
The index of a menu starts from one for the first present menu. If the index
is negative or zero, then the new menu is added before the first menu. If
the index exceeds the number of menus, then the new menu is added behind the
last menu.
*/

instance Menus (Menu m) | MenuElements m
instance Menus (PopUpMenu m) | PopUpMenuElements m

/*
PopUpMenus can only be opened in a SDI or MDI process. If the parent process is
a NDI process, then no PopUpMenu is opened and ErrorViolateDI is returned.
The elements of a PopUpMenu are the same as for standard menus except for
SubMenus. For elements the same restrictions hold as for standard menus.
The PopUpMenu will be closed as soon as the user has dismissed it either by
selecting an item or clicking outside the menu.
*/

closeMenu :: Id !(IOSt .1) -> IOSt .1

/*
closeMenu closes the indicated Menu and all of its elements.
The WindowMenu can not be closed by closeMenu (in case the Id argument equals
WindowMenuId).
*/

openMenuElements :: Id !Index .ls (m .ls (PSt .1)) !(PSt .1)
                    -> ((ErrorReport, PSt .1)) | MenuElements m
openSubMenuElements :: !Id !Index .ls (m .ls (PSt .1)) !(PSt .1)
                      -> ((ErrorReport, PSt .1)) | MenuElements m
openRadioMenuItems :: !Id !Index !MenuRadioItem (PSt .1) !(IOSt .1)
                     -> ((ErrorReport, IOSt .1))

/*
Add menu elements to the indicated Menu, SubMenu, or RadioMenu.
openRadioMenuItems checks the first item in the list if the RadioMenu was empty.
*/
Menu elements are added after the item with the specified index. The index of a menu element starts from one for the first menu element in the indicated menu.

If the index is negative or zero, then the new menu elements are added before the first menu element of the indicated menu.

If the index exceeds the number of menu elements in the indicated menu, then the new menu elements are added behind the last menu element of the indicated menu.

No menu elements are added if the indicated menu does not exist. In this case 
ErrorUnknownObject is returned.

open(Sub)MenuElements have no effect in case menu elements with duplicate Ids are opened. In this case ErrorIdsInUse is returned.

```
closeMenuElements :: !Id ![Id] (!$Sel .1) -> $Sel .1
/* closeMenuElements
   closes menu elements of the Menu identified by the first Id argument by their Ids. The elements of (Sub/Radio)Menus will be removed first.
*/
```

```
closeMenuIndexElements :: !Id ![Index] (!$Sel .1) -> $Sel .1
closeSubMenuIndexElements :: !Id ![Index] (!$Sel .1) -> $Sel .1
closeRadioMenuIndexElements :: !Id ![Index] (!$Sel .1) -> $Sel .1
/* Close menu elements of the indicated Menu, SubMenu, or RadioMenu by their Index position.
   Analogous to openMenuElements and openRadioMenuItems indices range from one to the number of menu elements in a menu. Invalid indices (less than one or larger than the number of menu elements of the menu) are ignored.
   If the currently checked element of a RadioMenu is closed, the first remaining element of that RadioMenu will be checked.
   Closing a (Sub/Radio)Menu closes the indicated (Sub/Radio)Menu and all of its elements.
*/
```

```
enableMenuSystem :: (!$Sel .1) -> $Sel .1
disableMenuSystem :: (!$Sel .1) -> $Sel .1
/* Enable/disable the menu system of this interactive process. When the menu system is re-enabled the previously selectable menus and elements will become selectable again.
   Enable/disable operations on the menu(elements) of a disabled menu system take effect when the menu system is re-enabled.
   enableMenuSystem has no effect in case the interactive process has a (number of) modal dialogue(s).
*/
```

```
enableMenus :: ![Id] (!$Sel .1) -> $Sel .1
disableMenus :: ![Id] (!$Sel .1) -> $Sel .1
/* Enable/disable individual menus.
   The WindowMenu can not be enabled/disabled.
   Disabling a menu overrules the SelectStates of its elements, which become unselectable.
   Enabling a disabled menu re-establishes the SelectStates of its elements.
   Enable/disable operations on the elements of a disabled menu take effect when the menu is re-enabled.
*/
```

```
getMenuSelectState :: !Id (!$Sel .1) -> ($Maybe SelectState, !$Sel .1)
/* getMenuSelectState yields the current SelectState of the indicated menu. In case the menu does not exist, Nothing is returned.
*/
```
getMenus :: !(integer!list) -> (![(integer!string)],integer!list)
/* getMenus yields the IDs and MenuTypes of the current set of menus of this
interactive process. */

getMenuPos :: !integer!list -> (!maybe!integer!list)
/* getMenuPos yields the index position of the indicated menu in the current list
of menus.
In case the menu does not exist, Nothing is returned. */

setMenuTitle :: !integer!string!list -> integer!list
getMenuTitle :: !integer!list -> (!maybe!string!list)
/* setMenuTitle sets the title of the indicated menu.
   In case the menu does not exist or refers to the WindowMenu, nothing
   happens.
getMenuTitle retrieves the current title of the indicated menu.
   In case the menu does not exist, Nothing is returned. */
A.18 StdMenuAttribute

definition module StdMenuAttribute

// Clean Standard Object I/O library, version 1.2
// StdMenuAttribute specifies which MenuAttributes are valid for each of the
// standard menus and menu elements.
// Basic comparison operations and retrieval functions are also included.
import StdMenuDef

/* The following functions specify the valid attributes for each standard menu
   (element).
*/
isValidMenuAttribute :: !(MenuAttribute .st) -> Bool
/* Menu
   (y /= valid, /= invalid)
   MenuFunction . | MenuInit   y | MenuSelectState y |
   MenuId       y | MenuMarkState . | MenuShortKey   . |
   MenuIndex    y | MenuModsFunction . |
*/
isValidSubMenuAttribute :: !(MenuAttribute .st) -> Bool
/* SubMenu
   (y /= valid, /= invalid)
   MenuFunction . | MenuInit   . | MenuSelectState y |
   MenuId       y | MenuMarkState . | MenuShortKey   . |
   MenuIndex    . | MenuModsFunction . |
*/
isValidRadioMenuAttribute :: !(MenuAttribute .st) -> Bool
/* RadioMenu
   (y /= valid, /= invalid)
   MenuFunction . | MenuInit   . | MenuSelectState y |
   MenuId       y | MenuMarkState . | MenuShortKey   . |
   MenuIndex    . | MenuModsFunction . |
*/
isValidMenuItemAttribute :: !(MenuAttribute .st) -> Bool
/* MenuItem
   (y /= valid, /= invalid)
   MenuFunction y | MenuInit   . | MenuSelectState y |
   MenuId       y | MenuMarkState . | MenuShortKey   y |
   MenuIndex    . | MenuModsFunction y |
*/
isValidMenuSeparatorAttribute :: !(MenuAttribute .st) -> Bool
/* MenuSeparator
   (y /= valid, /= invalid)
   MenuFunction . | MenuInit   . | MenuSelectState  |
   MenuId       y | MenuMarkState . | MenuShortKey   . |
   MenuIndex    . | MenuModsFunction . |
*/

/* The following functions return True only iff the attribute equals the
   indicated name.
*/
isMenuFunction   :: !(MenuAttribute .st) -> Bool
isMenuId        :: !(MenuAttribute .st) -> Bool
isMenuIndex     :: !(MenuAttribute .st) -> Bool
isMenuInit     :: !(MenuAttribute .st) -> Bool
isMenuMarkState :: !(MenuAttribute .st) -> Bool
isMenuModsFunction :: !(MenuAttribute .st) -> Bool
isMenuSelectState :: !(MenuAttribute .st) -> Bool
isMenuShortKey :: !MenuAttribute .st) -> Bool

/* The following functions return the attribute value if appropriate.
   THESE ARE PARTIAL FUNCTIONS! They are only defined on the corresponding
   attribute.
*/
getMenuFun :: !MenuAttribute .st) -> IdFun .st
getMenuIdAtt :: !MenuAttribute .st) -> Id
getMenuIndexAtt :: !MenuAttribute .st) -> Index
getMenuInitFun :: !MenuAttribute .st) -> IdFun .st
getMenuMarkStateAtt :: !MenuAttribute .st) -> MarkState
getMenuModsFun :: !MenuAttribute .st) -> ModifiersFunction .st
getMenuSelectStateAtt :: !MenuAttribute .st) -> SelectState
getMenuShortKeyAtt :: !MenuAttribute .st) -> Char
A.19 StdMenuDef

definition module StdMenuDef

// *************************************************
// Clean Standard Object I/O library, version 1.2
// StdMenu contains the types to define the standard set of menus and their
// elements.
// *************************************************

import StdIOCommon, StdMaybe

/* Menus: */
:: Menu   ls pst = Menu  Title    (m ls pst)
            [MenuAttribute *(ls,pst)]
:: PopUpMenu   ls pst = PopUpMenu   (m ls pst)

/* Menu elements: */
:: MenuItem ls pst = MenuItem  Title
            [MenuAttribute *(ls,pst)]
:: MenuSeparator ls pst = MenuSeparator [MenuAttribute *(ls,pst)]
:: RadioMenu ls pst = RadioMenu [MenuRadioItem *(ls,pst)] Index
:: SubMenu m ls pst = SubMenu Title    (m ls pst)
            [MenuAttribute *(ls,pst)]

:: MenuRadioItem st := (Title,Maybe Id,Maybe Char,IdFun st)

:: MenuAttribute st
   // Default:
   // Attributes for Menus and MenuElements:
   | MenuId       Id        // no Id
   | MenuSelectState      SelectState // menu(item) Able
   // Attributes only for Menus:
   | MenuIndex       Int      // end of current menu list
   | MenuInit        (IdFun st)  // no actions after opening menu
   // Attributes ignored by (Sub)Menus:
   | MenuFunction        (IdFun st)  // u->x
   | MenuMarkState      MarkState // NoMark
   | MenuModifiersFunction (ModifiersFunction st) // MenuFunction
   | MenuShortKey    Char      // no ShortKey
:: MenuType  := String
:: MenuElementType := String
A.20 StdMenuElement

definition module StdMenuElement

//*******************************************************************************
// Clean Standard Object I/O library, version 1.2
// StdMenuElement specifies all functions on menu elements.
//*******************************************************************************

import StdMenuDef
from iostate import IOSt

/*@ Functions that change the state of menu elements. 
Only those Id arguments that refer to menu elements within the same interactive 
process are used to change the corresponding menu elements. */

enableMenuElements :: ![Id] !(IOSt .1) -> IOSt .1
disableMenuElements :: ![Id] !(IOSt .1) -> IOSt .1

/*@ (en/dis)ableMenuElements set the SelectState of the indicated menu elements. 
Disabling a (Sub/Radio)Menu overrules the SelectStates of its elements, which 
become unselectable. 
Enabling a disabled (Sub/Radio)Menu re-establishes the SelectStates of its 
elements. 
(En/Dis)able operations on the elements of a disabled (Sub/Radio)Menu take 
effect when the (Sub/Radio)Menu is re-enabled. */

setMenuElementTitles :: ![Id,Title] !(IOSt .1) -> IOSt .1

/*@ setMenuElementTitles sets the titles of the indicated menu elements. */

markMenuItems :: ![Id] !(IOSt .1) -> IOSt .1
unmarkMenuItems :: ![Id] !(IOSt .1) -> IOSt .1

/*@ (un)markMenuItems sets the MarkState of the indicated MenuItems. */

selectRadioMenuItem :: ![Id] !(IOSt .1) -> IOSt .1
selectRadioMenuIndexItem :: ![Id,Index] !(IOSt .1) -> IOSt .1

/*@ selectRadioMenuItem selects the indicated MenuRadioItem of a RadioMenu, causing the mark of the 
previously marked MenuRadioItem to disappear. 
selectRadioMenuIndexItem indicates the MenuRadioItem by the Id of its parent RadioMenu and its Id. 
selectRadioMenuIndexItem indicates the MenuRadioItem by the Id of its parent RadioMenu and its index 
position (counted from 1). */

/*@ Access functions on MState. To read the state of a menu element, a MState is 
required which can be obtained by the getMenu function. The MState value 
represents the state of a menu at that particular moment. */

:: MState

getMenu :: ![Id] !(IOSt .1) -> ![Maybe MState, IOSt .1)
getParentMenu :: ![Id] !(IOSt .1) -> ![Maybe MState, IOSt .1)

/*@ getMenu returns a read-only MState for the indicated menu. 
In case the indicated menu does not exist Nothing is returned. 
getParentMenu returns a read-only MState for the indicated menu element. 
In case the Id does not correspond with a menu element, then Nothing
is returned.

*/

getMenuElementTypes :: !MState -> [(MenuElementType, Maybe Id)]
getCompoundMenuElementTypes :: !Id !MState -> [(MenuElementType, Maybe Id)]

/*

yields the list of MenuElementTypes of all menu elements of this menu.

getCompoundMenuElementTypes

yields the list of MenuElementTypes of all menu elements of this
(Sub/Radio)Menu.

Both functions return (Just id) if the element has a MenuId attribute, and
Nothing otherwise.

Ids are not collected recursively through (Sub/Radio)Menus.

*/

/* Functions that return the current state of menu elements.

For each access there is one singular and one plural version. In case of the
plural version the result list is of equal length as the argument Id list. Each
result list element corresponds in order with the argument Id list.

In both versions the first Boolean result is False in case of invalid Ids (if so
dummy values are returned - see comment).

Important: menu elements with no MenuId attribute, or illegal ids, can not be
found in the MState!

*/

getSelectedRadioMenuItems :: ![Id] !MState -> [[!Index, Maybe Id]]
getSelectedRadioMenuItems :: ![Id] !MState -> [[!Index, Maybe Id]]

/*

getSelectedRadioMenuItems(s)

returns the Index and Id, if any, of the currently selected MenuRadioItem of
the indicated RadioMenu.

If the RadioMenu does not exist or is empty, the Index is zero and the Id is
Nothing.

*/

getMenuElementSelectStates :: ![Id] !MState -> [(Bool, SelectState)]
getMenuElementSelectState :: ![Id] !MState -> (Bool, SelectState)

/*

getMenuElementSelectState(s)

yields the SelectStates of the indicated elements.

If the element does not exist Able is returned.

*/

getMenuElementMarkStates :: ![Id] !MState -> [(Bool, MarkState)]
getMenuElementMarkState :: ![Id] !MState -> (Bool, MarkState)

/*

getMenuElementMarkState(s)

yields the MarkState of the indicated elements.

If the element does not exist NoMark is returned.

*/

getMenuElementTitles :: ![Id] !MState -> [(Bool, Maybe String)]
getMenuElementTitle :: ![Id] !MState -> (Bool, Maybe String)

/*

getMenuElementTitle(s)

yields (Just title) of the indicated (SubMenu/MenuItem),
Nothing otherwise.

*/

getMenuElementShortKeys :: ![Id] !MState -> [(Bool, Maybe Char)]
getMenuElementShortKey :: ![Id] !MState -> (Bool, Maybe Char)

/*

getMenuElementShortKey(s)

yields (Just key) of the indicated MenuItem, Nothing
otherwise.

*/
A.21 StdMenuElementClass

definition module StdMenuElementClass

// ************************************************************************
// Clean Standard Object I/O library, version 1.2
// StdMenuElementClass defines the standard set of menu element instances.
// ************************************************************************

import StdMenuDef from StdPSt import PSt, PSt
from menuhandle import MenuElementState

/* Menu elements for Menus:
*/
class MenuElements m where
    menuElementToHandles :: !m ls (PSt .1)) ![PSt .1)
        -> ![MenuElementState .ls (PSt .1). ![PSt .1)
    getMenuElementType :: !m .pst)
        -> MenuElementType
instance MenuElements (AddLS m) | MenuElements m
instance MenuElements (NewLS m) | MenuElements m
instance MenuElements (ListLS m) | MenuElements m
instance MenuElements NilLS
instance MenuElements (::{*} m1 m2) | MenuElements m1
instance MenuElements (SubMenu m) | MenuElements m
instance MenuElements MenuItem
instance MenuElements MenuSeparator

/* Menu elements for PopUpMenus:
*/
class PopUpMenuElements m where
    popUpMenuElementToHandles :: !m ls (PSt .1)) ![PSt .1)
        -> ![MenuElementState .ls (PSt .1). ![PSt .1)
    getPopUpMenuElementType :: !m .pst)
        -> MenuElementType
instance PopUpMenuElements (AddLS m) | PopUpMenuElements m
instance PopUpMenuElements (NewLS m) | PopUpMenuElements m
instance PopUpMenuElements (ListLS m) | PopUpMenuElements m
instance PopUpMenuElements NilLS
instance PopUpMenuElements (::{*} m1 m2) | PopUpMenuElements m1
instance PopUpMenuElements RadioMenu
instance PopUpMenuElements MenuItem
instance PopUpMenuElements MenuSeparator
A.22 StdMenuReceiver

definition module StdMenuReceiver

    // ******************************************************************************
    // Clean Standard Object I/O library, version 1.2
    //
    // StdMenuReceiver defines Receiver(2) menu element instances.
    // ******************************************************************************

    import StdReceiverDef, StdMenuElementClass

    // Receiver components:
    instance MenuElements (Receiver m )
    instance MenuElements (Receiver2 m r)
A.23 StdPicture

definition module StdPicture

//*******************************************************************************
// Clean Standard Object I/O library, version 1.2
//
// StdPicture contains the drawing operations and access to Pictures.
//*******************************************************************************

from StdFunc import St
from osfont import Font
from ospicture import Picture
import StdPictureDef

// Pen attribute functions:

setPenAttributes :: Picture -> Picture
getPenAttributes :: Picture -> Picture
)[PenAttribute](Picture)

// Pen position attributes:

setPenPos :: Point2 -> Picture
getPenPos :: Picture -> (Point2, Picture)

class movePenPos figure :: Picture
// Move the pen position as much as when drawing the figure.
instance movePenPos Vector2
instance movePenPos Curve

// Pen size attributes:

setPenSize :: Int -> Picture
getPenSize :: Picture -> Int
setDefaultPenSize :: Picture -> Picture
// setDefaultPenSize = setPenSize 1

// Pen colour attributes:

setPenColour :: Colour -> Picture
getPenColour :: Picture -> Colour
getPenBack :: Picture -> Colour
setDefaultPenColour :: Picture -> Picture
setDefaultPenBack :: Picture -> Picture

toRGBColour :: Colour -> RGBColour // Convert a colour to RGBColour

// setDefaultPenColour = setPenColour Black
// setDefaultPenBack = setPenBack White

// Pen font attributes:

setPenFont :: Font -> Picture
getPenFont :: Picture -> Font
setDefaultPenFont :: Picture -> Picture

/* Font operations:
*/
openFont :: !FontDef -> Picture -> (!Bool, Font, Picture)
onDefaultFont :: !Picture -> (Font, !Picture)
onDialogFont :: !Picture -> (Font, !Picture)
/* openFont
creates the font as specified by the name, stylistic variations, and size. The Boolean result is True only if the font is available and need not be scaled. In all other cases, an existing font is returned (depending on the system).

openDefaultFont returns the font used by default by applications.
openDialogFont returns the font used by default by the system.

/*

getFontNames :: !Picture -> (![FontName], !Picture)
getFontStyles :: !FontName !Picture -> (![FontStyle], !Picture)
getFontSizes :: !Int !Int !FontName !Picture -> (![FontSize], !Picture)

*/

g.getFontNames returns the FontNames of all available fonts.
g.getFontStyles returns the FontStyles of all available styles of a particular FontName.
g.getFontSizes returns all FontSizes in increasing order of a particular FontName that are available without scaling. The sizes inspected are inclusive between the two Integer arguments. (Negative values are set to zero.)

In case the requested font is unavailable, the styles or sizes of the default font are returned.

/*
g.getFontDef :: !Font -> FontDef

*/
g.getFontDef returns the name, stylistic variations and size of the argument Font.

/*

g.getFontCharWidth :: !Char !Picture -> (!Int, !Picture)
g.getFontCharWidths :: !Char !Picture -> (![Int], !Picture)
g.getFontStringWidth :: !String !Picture -> (!Int, !Picture)
g.getFontStringWidths :: !String !Picture -> (![Int], !Picture)
g.getFontMetrics :: !Picture -> (!FontMetrics, !Picture)

g.getFontCharWidth returns the width of the argument (Char/String) given the Font argument or current PenFont attribute.
g.getFontCharWidths returns the widths of the argument (Char/String)s given the Font argument or current PenFont attribute.
g.getFontStringWidth returns the width of the argument String given the Font argument or current PenFont attribute.
g.getFontStringWidths returns the widths of the argument Strings given the Font argument or current PenFont attribute.
g.getFontMetrics returns the FontMetrics of the Font argument or current PenFont attribute.

/*

Region functions.
A Region is defined by a collection of shapes.

*/

class toRegion area :: !area -> Region

// Basic access functions on Regions:
inEmptyRegion :: !Region -> Bool
getRegionBound :: !Region -> Rectangle

*/

// Constructing a region:

class toRegion area :: !area -> Region
:: PolygonAt
   = { polygon_pos :: !Point2
      , polygon :: !Polygon
   }

instance toRegion Rectangle
instance toRegion PolygonAt
instance toRegion [area] | toRegion area
instance toRegion (::: area1 area2) | toRegion area1 & toRegion area2

// Drawing and restoring picture attributes:

appPicture :: !.(IdFun "Picture") !"Picture -> "Picture
accPicture :: !.(St "Picture .x") !"Picture -> (.x,"Picture)
/* (app/acc)Picture f pict
   apply f to pict. After drawing, the picture attributes of the result
   picture are restored to those of pict.
*/

// Drawing within a clipping region:

appClipPicture :: !Region !.(IdFun "Picture") !"Picture -> "Picture
accClipPicture :: !Region !.(St "Picture .x") !"Picture -> (.x,"Picture)

// Drawing in 'exclusive or' mode:

appXorPicture :: !.(IdFun "Picture") !"Picture -> "Picture
accXorPicture :: !.(St "Picture .x") !"Picture -> (.x,"Picture)
/* (app/acc)XorPicture f pict
   apply f to pict in the appropriate platform xor mode.
*/

// Drawing in 'hilite' mode:

class Hilites figure where
    hilite :: !figure !"Picture -> "Picture
    hiliteAt :: !Point2 !figure !"Picture -> "Picture
/* hilite
   draws figures in the appropriate 'hilite' mode at the current pen position.
   hiliteAt
   draws figures in the appropriate 'hilite' mode at the argument pen position.
   Both functions reset the 'hilite' mode after drawing.
*/
instance Hilites Box       // Hilite a box
instance Hilites Rectangle // Hilite a rectangle (note: hiliteAt pos r /= hilite r)

// Drawing points:

drawPoint :: !"Picture -> "Picture
drawPointAt :: !Point2 !"Picture -> "Picture
/* drawPoint
   plots a point at the current pen position p and moves to p+(vx=1,vy=0)
   drawPointAt
   plots a point at the argument pen position, but retains the pen position.
*/

// Drawing lines:

drawLineTo :: !Point2 !"Picture -> "Picture
drawLine :: !Point2 !Point2 !"Picture -> "Picture
/* drawLineTo */
   draws a line from the current pen position to the argument point which
   becomes the new pen position.
   drawLine
       draws a line between the two argument points, but retains the pen position.
*/

/* Drawing and filling operations. */
These functions are divided into the following classes:

**Drawables:**
- **draw**  'line-oriented' figures at the current pen position.
- **drawAt** 'line-oriented' figures at the argument pen position.
- **undraw** = appPicture (draw f o putPenColour background)
- **undrawAt x f** = appPicture (drawAt x f o putPenColour background)

**Fillables:**
- **fill**  'area-oriented' figures at the current pen position.
- **fillAt** 'area-oriented' figures at the argument pen position.
- **unfill** f = appPicture (fill f o putPenColour background)
- **unfillAt x f** = appPicture (fillAt x f o putPenColour background)

/*
class **Drawables** figure where
draw :|figure |Picture -> |Picture
drawAt :|Point2 |figure |Picture -> |Picture
undraw :|figure |Picture -> |Picture
undrawAt :|Point2 |figure |Picture -> |Picture
/*

class **Fillables** figure where
fill :|figure |Picture -> |Picture
fillAt :|Point2 |figure |Picture -> |Picture
unfill :|figure |Picture -> |Picture
unfillAt :|Point2 |figure |Picture -> |Picture
*/

/* Text drawing operations: */
// Text is always drawn with the baseline at the y coordinate of the pen.

instance **Drawables** Char
instance **Drawables** (#Char)

/* draw text:
   draws the text starting at the current pen position.
   The new pen position is directly after the drawn text including spacing.
   drawAt p text:
       draws the text starting at p.
*/

/* Line2 drawing operations: */
instance **Drawables** Line2
/* draw l:
   is equal to drawLine l line_end1 line_end2.
   drawAt p l:
       draw l
   None of these functions change the pen position.
*/

/* Vector2 drawing operations: */
instance **Drawables** Vector2
/* draw v:
   draws a line from the current pen position pen to pen+v.
   drawAt p v:
       draws a line from p to p+v.
*/

/* Oval drawing operations: */
An Oval o is a transformed unit circle
with horizontal radius $r_x$ o.oval_rx
vertical radius $r_y$ o.oval_ry

Let $(x, y)$ be a point on the unit circle:
then $(x', y') = (x*r_x, y*r_y)$ is a point on o.

Let $(x, y)$ be a point on o:
then $(x', y') = (x/r_x, y/r_y)$ is a point on the unit circle.

*/
instance Drawables Oval
instance Fillables Oval

/* draw o:
 draws an oval with the current pen position being the center of the oval.
 drawAt p o:
 draws an oval with p being the center of the oval.
 fill o:
 fills an oval with the current pen position being the center of the oval.
 fillAt p o:
 fills an oval with p being the center of the oval.

None of these functions change the pen position.
*/

/* Curve drawing operations:
A Curve c is a slice of an oval o
with start angle $a$ c.curve_from
end angle $b$ c.curve_to
direction $d$ c.curve_clockwise
The angles are taken in radians (counter-clockwise).
If $d$ holds then the drawing direction is clockwise, otherwise drawing occurs
counter-clockwise.
*/
instance Drawables Curve
instance Fillables Curve

/* draw c:
 draws a curve with the starting angle $a$ at the current pen position.
The pen position ends at ending angle $b$.
 drawAt p c:
 draws a curve with the starting angle $a$ at p.
 fill c:
 fills the figure obtained by connecting the endpoints of the drawn curve
(draw c) with the center of the curve oval.
The pen position ends at ending angle $b$.
 fillAt p c:
 fills the figure obtained by connecting the endpoints of the drawn curve
(drawAt p c) with the center of the curve oval.
*/

/* Box drawing operations:
A Box b is a horizontally oriented rectangle
with width $w$ b.box_w
height $h$ b.box_h
In case $w=0$ ($h=0$), the Box collapses to a vertical (horizontal) vector.
In case $w=0$ and $h=0$, the Box collapses to a point.
*/
instance Drawables Box
instance Fillables Box

/* draw b:
 draws a box with left-top corner at the current pen position p and
right-bottom corner at $p+(w,h)$.
 drawAt p b:
 draws a box with left-top corner at p and right-bottom corner at $p+(w,h)$.
 fill b:
 fills a box with left-top corner at the current pen position p and
right-bottom corner at $p+(w,h)$.
 fillAt p b:
 fills a box with left-top corner at p and right-bottom corner at $p+(w,h)$.

None of these functions change the pen position.
/*

// Rectangle drawing operations:
A Rectangle r is always horizontally oriented
with width w = abs (r.corner1.x - r.corner2.x)
height h = abs (r.corner1.y - r.corner2.y)
In case w = 0 (h = 0), the Rectangle collapses to a vertical (horizontal) vector.
In case w = 0 and h = 0, the Rectangle collapses to a point.

*/
instance Drawables Rectangle
instance Fillables Rectangle
/*
draw r:
draws a rectangle with diagonal corners r.corner1 and r.corner2.
drawAt p r:
draw r
fill r:
fills a rectangle with diagonal corners r.corner1 and r.corner2.
fillAt p r:
fill r
None of these functions change the pen position.

*/

/*

// Polygon drawing operations:
A Polygon p is a figure
with shape p.polygon_shape
A polygon p at a point base is drawn as follows:
drawPicture [setPenPos base:map draw shape] ++ [drawToPoint base]

*/
instance Drawables Polygon
instance Fillables Polygon
/*  None of these functions change the pen position.

*/

getResolution :: !Picture -> (!!(Int, Int), !Picture)
/*  getResolution returns the horizontal and vertical resolution of a Picture in dpi
    (dots per inch).
  In case of a printer Picture:
    the return values are the printer resolution (if it is not emulating the
    screen resolution).
  In case of a screen Picture:
    the resolution is the number of pixels that fit in one "screen inch".
    A "screen inch" is the physical size of a 72 point font on the screen.
    Although some screens allow the user to alter the screen resolution,
    getResolution always returns the same value for the same screen.
    The reason is that if a user for instance increases the screen resolution,
    not only the size of a pixel decreases, but also the size of a "screen
    inch". So a 12 point font will not appear with 12 point size, but smaller
    (A point is a physical unit, defined as 1 point is approximately 1/72 inch.)

*/
A.24 StdPictureDef

```haskell
definition module StdPictureDef

// Clean Standard Object I/O library, version 1.2
// StdPictureDef contains the predefined figures that can be drawn.

import StdIOBasic
from osfont import Font

:: Line2 // A line connects two points
  = { line_end1 :: !Point2 // The first point
     , line_end2 :: !Point2 // The second point
     }

:: Box // A box is a rectangle
  = { box_w :: !Int // The width of the box
     , box_h :: !Int // The height of the box
     }

:: Oval // An oval is a stretched unit circle
  = { oval_rx :: !Int // The horizontal radius (stretch)
     , oval_ry :: !Int // The vertical radius (stretch)
     }

:: Curve // A curve is a slice of an oval
  = { curve_oval :: !Oval // The source oval
     , curve_from :: !Real // Starting angle (in radians)
     , curve_to :: !Real // Ending angle (in radians)
     , curve_clockwise :: !Bool // Direction: True iff clockwise
     }

:: Polygon // A polygon is an outline shape
  = { polygon_shape :: ![Vector2] // The shape of the polygon
     }

:: FontDef
  = { fName :: !FontName // Name of the font
      , fStyles :: ![FontStyle] // Stylistic variations
      , fSize :: ![FontSize] // Size in points
      }

:: FontMetrics
  = { fAscent :: !Int // Distance between top and base line
      , fDescent :: !Int // Distance between bottom and base line
      , fLeading :: !Int // Distance between two text lines
      , fMaxWidth :: !Int // Max character width including spacing
      }

:: FontName := String
:: FontStyle := String
:: FontSize := Int
:: Colour
  = RGBA RGBAColour
    | Black | White | DarkGrey | Grey | LightGrey // 75%, 50%, and 25% Black
    | Red   | Green | Blue    |
    | Cyan  | Magenta | Yellow
  = RGBA RGBAColour
    | r :: !Int // The contribution of red
    , g :: !Int // The contribution of green
    , b :: !Int // The contribution of blue
  }

:: PenAttribute // Default:
  = PenSize Int // 1
  | PenPos Point2 // zero
  | PenColour Colour // Black
  | PenBack Colour // White
```
A.24. STDPICTUREDEF

| PenFont | Font    // DefaultFont |

// Colour constants:
BlackRGB := \{r=MinRGB,g=MinRGB,b=MinRGB\}
WhiteRGB := \{r=MaxRGB,g=MaxRGB,b=MaxRGB\}
MinRGB := 0
MaxRGB := 255

// Font constants:
SerifFontDef := \{fName="Times New Roman", fStyles=[], fSize=10\}
SansSerifFontDef := \{fName="Arial", fStyles=[], fSize=10\}
SmallFontDef := \{fName="Small Fonts", fStyles=[], fSize=7\}
NonProportionalFontDef := \{fName="Courier New", fStyles=[], fSize=10\}
SymbolFontDef := \{fName="Symbol", fStyles=[], fSize=10\}

// Font style constants:
ItalicsStyle := "Italic"
BoldStyle := "Bold"
UnderlinedStyle := "Underline"

// Standard lineheight of a font is the sum of its leading, ascent and descent:
fontLineHeight fMetrics := fMetrics.fLeading + fMetrics.fAscent + fMetrics.fDescent

// Useful when working with Ovals and Curves:
PI := 3.1415926535898
A.25 StdPrint

Cancelled, StartedPrinting, PrintEnvironments

definition module StdPrint

// Clean Standard Object I/O library, version 1.2
// StdPrint specifies general printing functions.
// **********************************************************************

from StdIOCommon import UpdateState, ViewFrame, UpdateArea
from StdIBasic import IdFun, Size, Rectangle, Point2
from StdOverloaded import !==
from ospicture import Picture
from osprint import PrintSetup, JobInfo, PrintInfo, Alternative,
  Cancelled, StartedPrinting, PrintEnvironments
from iostate import IOSt, PST
from StdFile import FileEnv, Files

:: PageDimensions
  - { page : Size // Size of the drawable area of the page
       , margins : Rectangle // This field contains information about the
         size of the margins on a sheet in pixels.
         // Drawing can't occur within these margins.
         // The margin Rectangle is bigger than the
         // page size. Its values are:
         // corner1.x<=0 k & corner1.y<0 k
         // corner2.x>=page.w k & corner2.y>=page.h
       , resolution : (Int, Int) // Horizontal and vertical printer
         // resolution in dpi
     }

defaultPrintSetup : !env -> (!PrintSetup, !env)
  | FileEnv env
  /* defaultPrintSetup returns a default print setup. */

printSetupDialog : !PrintSetup !+env -> (!PrintSetup, !+env)
  | PrintEnvironments env
  /* printSetupDialog lets the user choose a print setup via the print setup dialog. */

getPageDimensions : !PrintSetup !Bool -> PageDimensions
  instance PageDimensions

fwritePrintSetup : !PrintSetup !+File -> !File
  /* fwritePrintSetup writes PrintSetup to file (text or data). */

freadPrintSetup : !File !+env -> (!Bool, !PrintSetup, !File, !+env)
  | FileEnv env
  /* freadPrintSetup reads PrintSetup from File (text or data).
   * If the resulting Boolean is True: success, otherwise the default PrintSetup is
   * returned. */

print : !Bool !Bool
  (PrintInfo !+Picture -> ([IdFun +Picture], !+Picture))
  !PrintSetup !+env
  -> !PrintSetup !+env
  | PrintEnvironments env
  /* printDoDialog emulateScreen pages printSetup env
   * sends output to the printer and returns the used print setup, which can differ
   * from the input print setup. */
doDialog:
  if True a dialog will pop up that lets the user choose all printing options, otherwise printing will happen in the default way.

emulateScreen:
  if True, the printing routine will emulate the resolution of the screen. That means that a pixel on paper has the same dimension as on screen. Otherwise, the used resolution will be the printer resolution, with the effect that coordinates get much "tighter".

pages:
  this function should calculate a list of functions, each function representing one page to be printed. Each of these drawing functions is applied to an initial printer Picture.

env:
  a PrintEnvironment is either the PST or the Files system.

printUpdateFunction
  :: !Bool (UpdateState -> *Picture -> *Picture) [Rectangle]
  !PrintSetup !*env
  -> (!PrintSetup,!*env)
  | PrintEnvironments env

page
  printUpdateFunction doDialog update area printSetup env
  sends the content of the update function of a given area to the printer:
  doDialog:
    identical to print.
  update:
    this function will be applied to an UpdateState of value
    {oldFrame=area,newFrame=area,updArea=[area]}.
  area:
    the area to be sent to the printer. If a rectangle of this area does not
    fit on one sheet, it will be distributed on several sheets.
  printSetup,env,result value:
    identical to print.

printPagePerPage
  :: !Bool !Bool
  .x
  .([x -> .(PrintInfo -> .(*Picture -> .(Bool,Point2,.(state,*Picture))))]
  (.(state,*Picture) -> .(Bool,Point2,(state,*Picture)))
  !PrintSetup
  !*env
  -> (Alternative .x .state,!*env)
  | PrintEnvironments env

page
  printPagePerPage doDialog emulateScreen x prepare pages printSetup env
  sends output to the printer. This function can be used more efficiently than print. The major difference is
  that the pages function is a state transition function instead of a page list producing function. Each page transition function generates one page for the printer. An additional feature of printPagePerPage is that it is possible to set the origin of the printer Pictures.
  doDialog:
    identical to print.
  emulateScreen:
    identical to print.
  x:
    this value is passed to the prepare function.
  prepare:
    this function calculates the initial page print state.
    If there are no pages to print, the return Boolean must be True.
    The returned Point is the Origin of the first printer Picture.
  pages:
    this state transition function produces the printed pages.
    The state argument consists of the state information and an initial printer Picture which Origin has been set by the previous return Point value.
    If there are no more pages to print, the return Boolean must be True. In
    that case the result of printPagePerPage is (StartedPrinting state),
    with state the current state value. If printing should continue, the
return Boolean is False.
The returned Point is the Origin of the next printer Picture.
printSetup.env:
    identical to print.
    
    If printing is cancelled via the print dialog, then (Cancelled x) will be
    returned, otherwise (StartedPrinting ...)
*/

instance PrintEnvironments World
/* Other instances are the Files subworld and PSt.
*/
A.26 StdPrintText

definition module StdPrintText

// ******************************************************************************
// Clean Standard Object I/O library, version 1.2
//
// StdPrintText specifies functions to print text.
// ******************************************************************************

from StdPictureDef import FontDef, FontName, FontStyle, FontSize
import StdPrint
from StdString import String

:: WrapMode :== Int

NoWrap :== 0
LeftJustify :== 1
RightJustify :== 2

class CharStreams cs where
getchar :: ![cs -> (![Bool, Char, ![cs])
    // getChar returns the next character of the stream. The Boolean
    // result indicates whether this operation was successful.
savePos :: ![cs -> *cs
    // savePos saves actual position of charstream to enable the
    // restorePos function to restore it.
restorePos :: ![cs -> *cs
eos :: ![cs -> (![Bool, ![cs])
    // eos checks for end of stream.

instance CharStreams FileCharStream

:: *FileCharStream

fileToCharStream :: ![File -> *FileCharStream
charStreamToFile :: ![FileCharStream -> *File

printText1 :: ![Bool, !WrapMode, !FontDef, !Int
    ![charStream !PrintSetup ![env
    -> (![charStream, !PrintSetup, ![env)
    | CharStreams charStream & PrintEnvironments env
/* printText1 doDialog wrapMode font spacesPerTab
    charStream printSetup env
    prints a CharStream:
    doDialog:
        identical to print (StdPrint)
    wrapMode:
        controls word wrapping in case lines do not fit. NoWrap suppresses wrapping.
        LeftJustify and RightJustify wrap text to the left and right respectively.
    font:
        the text will be printed in this font.
    spacesPerTab:
        the number of spaces a tab symbol represents.
    charStream:
        the charStream to be printed.
    printSetup, env:
        identical to print (StdPrint)
*/

printText2 :: ![String, !String, !Bool, !WrapMode, !FontDef, !Int
    ![charStream !PrintSetup ![env
    -> (![charStream, !PrintSetup, ![env)
    | CharStreams charStream & PrintEnvironments env
printText2 titleStr pageStr
doDialog wrapMode fontParams spacesPerTab
charStream printSetup env
prints a charStream with a header on each page.
titleStr:
this String will be printed on each page at the left corner of the header
pageStr:
this String and the actual page number are printed on the right corner of
the header
The other parameters are identical to printText1.
*/

printText3 :: !Bool !WrapMode !FontDef !Int
  (PrintInfo +Picture -> (state, (Int,Int), +Picture))
  (state Int +Picture -> +Picture)
  !+charStream !PrintSetup !*env
  -> (!(!+charStream, !PrintSetup), !*env)
| CharStreams charStream & PrintEnvironments env
/* printText3 doDialog wrapMode font spacesPerTab
  textRange
  eachPageDraw
  charStream printSetup env
prints a charStream with a header and trailer on each page.
textRange:
this function takes a PrintInfo record and the printer Picture on which the
text will be printed. It returns a triple (state,range,picture):
state:
a value of arbitrary type that can be used to pass data to the page
printing function pages.
rangle:
a pair (top,bottom), where top<bottom. The printed text will appear
within these y-coordinates only, so a header and a trailer can be
printed for each page.
eachPageDraw:
this function draws the header and/or trailer for the current page. Its
arguments are the data produced by textRange, the actual page number,
and an initial printer Picture. This function is applied by printText3 before
each new page receives its text.
The other parameters are identical to printText1.
*/

/* If a file is opened with FReadData, then all possible newline conventions
(unix,mac,dos) will be recognized. All these printing functions will replace
nonprintable characters of the font with ASCII spaces. Exceptions are: newline,
formfeed and tab. So the ASCII space has to be a printable character in the used
font. A form feed character will cause a form feed, and it will also end a line.
*/
A.27 StdProcess

definition module StdProcess

// Clean Standard Object I/O library, version 1.2
// StdProcess contains the process creation and manipulation functions.

import StdProcessDef from iostate import PSt, IOSt

/* General process topology creation functions: */

class Processes pdef where
    startProcesses :: !pdef !World -> !World
    openProcesses :: !pdef !(PSt .1) -> PSt .1
/* (start/open)Processes creates an interactive process topology specified by
    the pdef argument.
    All interactive processes can communicate with each other by means of the
    file system or by message passing.
    startProcesses terminates as soon as all interactive processes that are
    created by startProcesses and their child processes have terminated.
    openProcesses schedules the interactive processes specified by the pdef argument
    to be created.
*/

instance Processes [pdef] | Processes pdef
instance Processes Process

startIO :: !DocumentInterface .1 !(ProcessInit (PSt .1))
      !(ProcessAttribute (PSt .1))
     !World -> !World
/* startIO creates one process group of one interactive process.
*/

/* Process access operations: */

closeProcess :: !(PSt .1) -> PSt .1
/* closeProcess removes all abstract devices that are held in the interactive
    process.
    If the interactive process has processes that share its GUI then these will also
    be closed recursively. As a result evaluation of this interactive process
    including GUI sharing processes will terminate.
*/

hideProcess :: !(PSt .1) -> PSt .1
showProcess :: !(PSt .1) -> PSt .1
/* If the interactive process is active, hideProcess hides the interactive process,
    and showProcess makes it visible. Note that hiding an interactive process does
    NOT disable the process but simply makes it invisible.
*/

getProcessWindowPos :: !(IOSt .1) -> (!Point2, !IOSt .1)
/* getProcessWindowPos returns the current position of the ProcessWindow.
*/

getProcessWindowSize :: !(IOSt .1) -> (!Size, !IOSt .1)
/* getProcessWindowSize returns the current size of the ProcessWindow.
*/
A.28 StdProcessAttribute

definition module StdProcessAttribute

// *************************************************
// Clean Standard Object I/O library, version 1.2
// StdProcessAttribute specifies which ProcessAttributes are valid for each of the
// standard interactive processes.
// Basic comparison operations and retrieval functions are also included.
// *************************************************

import StdProcessDef

/* The following function specifies the valid attributes for each standard
interactive process, specialised by its DocumentInterface. */

isProcessKindAttribute :: !DocumentInterface !(ProcessAttribute .st) -> Bool
/* (The document interface is given for which the attribute is valid)
ProcessActivate NDI SDI MDI | ProcessToolbar SDI MDI
ProcessClose NDI SDI MDI | ProcessWindowPos SDI MDI
ProcessDeactivate NDI SDI MDI | ProcessWindowSize SDI MDI
ProcessNoWindowMenu MDI | ProcessWindowResize SDI MDI
ProcessOpenFiles SDI MDI */

/* The following functions return True only iff the attribute equals the
indicated name. */

isProcessActivate :: !(ProcessAttribute .st) -> Bool
isProcessClose :: !(ProcessAttribute .st) -> Bool
isProcessDeactivate :: !(ProcessAttribute .st) -> Bool
isProcessNoWindowMenu :: !(ProcessAttribute .st) -> Bool
isProcessOpenFiles :: !(ProcessAttribute .st) -> Bool
isProcessToolbar :: !(ProcessAttribute .st) -> Bool
isProcessWindowPos :: !(ProcessAttribute .st) -> Bool
isProcessWindowSize :: !(ProcessAttribute .st) -> Bool

/* The following functions return the attribute value if appropriate.
THESE ARE PARTIAL FUNCTIONS! They are only defined on the corresponding
attribute. */

getProcessActivateFun :: !(ProcessAttribute .st) -> IdFun .st
getProcessCloseFun :: !(ProcessAttribute .st) -> IdFun .st
getProcessDeactivateFun :: !(ProcessAttribute .st) -> IdFun .st
getProcessOpenFilesFun :: !(ProcessAttribute .st) -> IdFun .st
-> ProcessOpenFilesFunction .st
getProcessToolbarAtt :: !(ProcessAttribute .st) -> [ToolbarItem .st]
getProcessWindowPosAtt :: !(ProcessAttribute .st) -> ItemPos
getProcessWindowSizeAtt :: !(ProcessAttribute .st) -> Size
getProcessWindowResizeFun :: !(ProcessAttribute .st) -> ProcessWindowResizeFunction .st


A.29  StdProcessDef

definition module StdProcessDef

// ******************************************************************************
// Clean Standard Object I/O library, version 1.2
//
// StdProcessDef contains the types to define interactive processes.
// ******************************************************************************

import StdIOCommon
from iostate import PST, ISt

:: Process = E . l: Process
    DocumentInterface // The process DocumentInterface
    l // The process private state
    [ProcessInit (PSt l)] // The process initialisation
    [ProcessAttribute (PSt l)] // The process attributes

/* MDI processes can't open windows and menus.
SDI processes can have at most one window open.
MDI processes can open an arbitrary number of device instances.
*/

:: ProcessInit pst
    ::= IdFun pst
A.30 StdPSt

definition module StdPSt

// ************************************************************************************
// Clean Standard Object I/O library, version 1.2
// StdPSt defines operations on PSt and IOS that are not abstract device related.
// ************************************************************************************

from StdFunc import St
from StdIOCommon import IdFun, DocumentInterface, MDI, SDI, NDI
from StdPicture import Picture
from iostate import PSt, IOS

/* accScreenPicture provides access to an initial Picture as it would be created in
a window or control. */

class accScreenPicture env :: !.(St *Picture .x) !*env -> (!.x,!*env)

instance accScreenPicture World
instance accScreenPicture (IOS .l)

beep :: !(IOS .l) -> IOS .l
/* beep emits the alert sound. */

// Operations on the DocumentInterface of an interactive process:

getDocumentInterface :: !(IOS .l) -> (!DocumentInterface, !IOS .l)
/* getDocumentInterface returns the DocumentInterface of the interactive process. */

// Operations on the attributes of an interactive process:

setProcessActivate :: !.(IdFun (PSt .l)) !(IOS .l) -> IOS .l
setProcessDeactivate:: !.(IdFun (PSt .l)) !(IOS .l) -> IOS .l
/* These functions set the ProcessActivate and ProcessDeactivate attribute of the
interactive process respectively. */

// Coercing PSt component operations to PSt operations.

appListPIO :: !.[.IdFun (IOS .l)] !(PSt .l) -> PSt .l
appListPLoc :: !.[.IdFun .l] !(PSt .l) -> PSt .l

appPIO :: !.(IdFun (IOS .l)) !(PSt .l) -> PSt .l
appLoc :: !.(IdFun .l) !(PSt .l) -> PSt .l

// Accessing PSt component operations.

accListPIO :: !.[.St (IOS .l) .x] !(PSt .l) -> (!.[x],!PSt .l)
accListPLoc :: !.[.St .l .x] !(PSt .l) -> (!.[x],!PSt .l)

accPIO :: !.(St (IOS .l) .x) !(PSt .l) -> (!.x, !PSt .l)
accLoc :: !.(St .l .x) !(PSt .l) -> (!.x, !PSt .l)
A.31 StdPStClass

definition module StdPStClass

// ..............................................................
// Clean Standard Object I/O library, version 1.2
//
// StdPStClass collects (PSt .l) and (IOSt .l) class instances.
// ..............................................................

import StdFile, StdFileSelect, StdSound, StdTime
from iostate import PSt, IOSt

/* PSt is an environment instance of the following classes:
   - FileSystem (see StdFile)
   - FileEnv (see StdFile)
   - FileSelectEnv (see StdFileSelect)
   - TimeEnv (see StdTime)
   - playSoundFile (see StdSound)

   IOSt is also an environment instance of the classes FileEnv, TimeEnv */
instance FileSystem (PSt .l)
instance FileEnv (PSt .l), (IOSt .l)
instance FileSelectEnv (PSt .l)
instance TimeEnv (PSt .l), (IOSt .l)
instance playSoundFile (PSt .l)
A.32 StdReceiver

definition module StdReceiver

// Clean Standard Object I/O library, version 1.2
// StdReceiver specifies all receiver operations.

import StdReceiverDef, StdMaybe
from iostate import PSt, IOSt
from id import RId, RId2Id, RIdtoId, RIdtoId2Id,

// Open uni- and bi-directional receivers:
class Receivers rdef where
opensReceiver :: !ls (*rdef .rl (PSt .1)) (!PSt .1) /*!ErrorReport, !PSt .1) getReceiverType :: *(*rdef .rl pst) -> ReceiverType
/* opens the given receiver if no receiver currently exists with the given R(2)Id. The R(2)Id has to be used to send messages to this receiver.
getReceiverType
returns the type of the receiver (see also getReceivers).
*/

instance Receivers (Receiver msg)
instance Receivers (Receiver2 msg resp)

closeReceiver :: !Id (!(IOSt .1)) -> IOSt .1
/* closeReceiver closes the indicated uni- or bi-directional receiver.
Invalid Ids have no effect.
*/

getReceivers :: !(IOSt .1) -> (![Id,ReceiverType], !IOSt .1)
/* getReceivers returns the Ids and ReceiverTypes of all currently open uni- or bi-directional receivers of this interactive process.
*/

enableReceivers :: ![Id] !(IOSt .1) -> IOSt .1
disableReceivers :: ![Id] !(IOSt .1) -> IOSt .1
getReceiverSelectState :: !Id !(IOSt .1) -> (!Maybe SelectState, !IOSt .1)
/* (en/dis)ableReceivers
(en/dis)able the indicated uni- or bi-directional receivers.
Note that this implies that in case of synchronous message passing messages
can fail (see the comments of syncSend and syncSend2 below). Invalid Ids
have no effect.
getReceiverSelectState
yields the current SelectState of the indicated receiver. In case the
receiver does not exist, Nothing is returned.
*/

// Inter-process communication:

// Message passing status report:
SendReport :: SendOk
| SendUnknownReceiver
| SendUnableReceiver
| SendDeadlock
| OtherSendReport !String

instance SendReport
instance toString SendReport
asyncSend :: !(RId msg) msg !(PSt .1) -> (SendReport, !PSt .1)

/* asyncSend posts a message to the receiver indicated by the argument RId. In case the indicated receiver belongs to this process, the message is simply buffered. asyncSend is asynchronous: the message will at some point be received by the indicated receiver.

The SendReport can be one of the following alternatives:
- SendOk: No exceptional situation has occurred. The message has been sent. Note that even though the message has been sent, it cannot be guaranteed that the message will actually be handled by the indicated receiver because it might become closed, forever disabled, or flooded with synchronous messages.
- SendUnknownReceiver: The indicated receiver does not exist.
- SendUnableReceiver: Does not occur: the message is always buffered, regardless whether the indicated receiver is Able or Unable. Note that in case the receiver never becomes Able, the message will not be handled.
- SendDeadlock: Does not occur.
*/

syncSend :: !(RId msg) msg !(PSt .1) -> (SendReport, !PSt .1)

/* syncSend posts a message to the receiver indicated by the argument RId. In case the indicated receiver belongs to the current process, the corresponding ReceiverFunction is applied directly to the message argument and current process state. syncSend is synchronous: this interactive process blocks evaluation until the indicated receiver has received the message.

The SendReport can be one of the following alternatives:
- SendOk: No exceptional situation has occurred. The message has been sent and handled by the indicated receiver.
- SendUnknownReceiver: The indicated receiver does not exist.
- SendUnableReceiver: The receiver exists, but its ReceiverSelectState is Unable. Message passing is halted. The message is not sent.
- SendDeadlock: The receiver is involved in a synchronous, cyclic communication with the current process. Blocking the current process would result in a deadlock situation. Message passing is halted to circumvent the deadlock. The message is not sent.
*/

syncSend2 :: !(R2Id msg resp) msg !(PSt .1)

-> (!!(SendReport, Maybe resp), !PSt .1)

/* syncSend2 posts a message to the receiver indicated by the argument R2Id. In case the indicated receiver belongs to the current process, the corresponding Receiver2Function is applied directly to the message argument and current process state. syncSend2 is synchronous: this interactive process blocks until the indicated receiver has received the message.

The SendReport can be one of the following alternatives:
- SendOk: No exceptional situation has occurred. The message has been sent and handled by the indicated receiver. The response of the receiver is returned as well as (Just response).
- SendUnknownReceiver: The indicated receiver does not exist.
- SendUnableReceiver: The receiver exists, but its ReceiverSelect is Unable. Message passing is halted. The message is not sent.
- SendDeadlock: The receiver is involved in a synchronous, cyclic communication with the current process. Blocking the current process would result in a deadlock situation. Message passing is halted to circumvent the deadlock. The message is not sent.

In all other cases than SendOk, the optional response is Nothing. */
A.33 StdReceiverAttribute

definition module StdReceiverAttribute

// ************************************************************************
// Clean Standard Object I/O library, version 1.2
// StdReceiverAttribute specifies which ReceiverAttributes are valid for each of
// the standard receivers.
// Basic comparison operations and retrieval functions are also included.
// ************************************************************************

import StdReceiverDef

/* The following functions specify the valid attributes for each standard receiver. */
isValidReceiverAttribute :: !(ReceiverAttribute .st) -> Bool
/* Receiver (y = valid, . = invalid)
   ReceiverInit y | ReceiverSelectState y |
*/

isValidReceiver2Attribute :: !(ReceiverAttribute .st) -> Bool
/* Receiver2 (y = valid, . = invalid)
   ReceiverInit y | ReceiverSelectState y |
*/

/* The following functions return True only iff the attribute equals the
   indicated name. */
isReceiverInit :: !(ReceiverAttribute .st) -> Bool
isReceiverSelectState :: !(ReceiverAttribute .st) -> Bool
isReceiverConnectedReceivers :: !(ReceiverAttribute .st) -> Bool // MW11++

/* The following functions return the attribute value if appropriate.
   THESE ARE PARTIAL FUNCTIONS! They are only defined on the corresponding
   attribute. */
getReceiverInitFun :: !(ReceiverAttribute .st) -> IdFun .st
getReceiverSelectStateAtt :: !(ReceiverAttribute .st) -> SelectState
getReceiverConnectedReceivers :: !(ReceiverAttribute .st) -> [Id] // MW11++
A.34 StdReceiverDef

definition module StdReceiverDef

// ************************************************************
// Clean Standard Object I/O library, version 1.2
// StdReceiverDef contains the types to define the standard set of receivers.
// ************************************************************

import StdIOCommon

:: Receiver m ls pst = Receiver (RId m)  [ReceiverFunction m *(ls.pst)]
                   [ReceiverAttribute *(ls.pst)]
:: Receiver2 m r ls pst = Receiver2 (R2Id m r) [Receiver2Function m r *(ls.pst)]
                      [ReceiverAttribute *(ls.pst)]

:: ReceiverFunction m st := m -> st -> st
:: Receiver2Function m r st := m -> st -> (r, st)

:: ReceiverAttribute st     // Default:
  = ReceiverInit (IdFun st) // no actions after opening receiver
  | ReceiverSelectState SelectState // receiver Able
  | ReceiverConnectedReceivers [Id] // []
:: ReceiverType
  ::= String
A.35 StdSound

definition module StdSound

// **************************************************************************
// Clean Standard Object I/O library, version 1.2
//
// StdSound specifies sound playing functions.
// **************************************************************************

from StdString import String

class playSoundFile env :: !String → !env → (!Bool, !env)

/* playSoundFile filename
   opens the sound file at filename and plays it synchronously.
   The Boolean result indicates whether the sound file could be successfully
   played.
*/

instance playSoundFile World
A.36  StdStringChannels

definition module StdStringChannels

// ******************************************************************************
// Clean Standard Object I/O library, version 1.2
//
// StdStringChannels provides channel instances to send and receive Strings.
// These channels use their own protocol above TCP.
// ******************************************************************************

from StdString import String
import StdTCPDef, StdChannels, StdEventTCP
from StdReceiver import Receivers, ReceiverType, RId
from StdTCPChannels import SelectSend, SelectReceive, getNrOfChannels

/** If a string via a StringChannel is sent, then first the length of the string is
  then the string itself, e.g. sending the string "abc" will result in
  "3 abc\x0D"
*/

// ******************************************************************************
// StringChannels to receive
// ******************************************************************************

::  StringRChannel a
::  StringRChannel := StringRChannel_ String
::  StringRChannels = StringRChannels [StringRChannel]
::  StringChannelReceiver ls pst
   StringChannelReceiver
   (RId (ReceiveMsg String)) StringRChannel
   (ReceiverFunction (ReceiveMsg String) *(ls,pst))
   [ReceiverAttribute *(ls,pst)]

toStringRChannel :: TCP_RChannel -> StringRChannel

instance Receivers StringChannelReceiver
instance Receive StringRChannel_
instance closeRChannel StringRChannel_
instance MaxSize StringRChannel_

// ******************************************************************************
// StringChannels to send
// ******************************************************************************

::  StringSChannel a
::  StringSChannel := StringSChannel_ String
::  StringSChannels = StringSChannels [StringSChannel]

toStringSChannel :: TCP_SChannel -> StringSChannel

instance Send StringSChannel_

// For openSendNotifier, closeSendNotifier
instance accSChannel StringSChannel_

// For selectChannel
instance SelectSend StringSChannels
instance SelectReceive StringSChannels
instance getNrOfChannels StringSChannels
A.37 StdSystem

definition module StdSystem

// **************************************************************************************************************************
// Clean Standard Object I/O library, version 1.2
//
// StdSystem defines platform dependent constants and functions.
// **************************************************************************************************************************

import StdIOMBasic

// System dependencies concerning the file system.
dirseparator :: Char   // Separator between folder- and filenames in a pathname
homepath :: !String -> String
applicationpath :: !String -> String
/* dirseparator
   is the separator symbol used between folder- and filenames in a file path.
homepath
   prefixes the 'home' directory file path to the given file name.
applicationpath
   prefixes the 'application' directory file path to the given file name.
Use these directories to store preference/options/help files of an application.
*/
newlineChars :: !String
/* the newline characters in a textfile */
*/
printSetupTypical :: Bool

// System dependencies concerning the time resolution
ticksPerSecond :: Int
/* ticksPerSecond returns the maximum timer resolution per second. */
*/

// System dependencies concerning the screen resolution.
mmperinch := 25.4
hmm :: !Real -> Int
vmm :: !Real -> Int
hinch :: !Real -> Int
vinch :: !Real -> Int
/* h(mm/inch) convert millimeters/inches into pixels, horizontally.
v(mm/inch) convert millimeters/inches into pixels, vertically. */

maxScrollWindowSize :: Size
maxFixedWindowSize :: Size
/* maxScrollWindowSize
   yields the range at which scrollbars are inactive.
maxFixedWindowSize
   yields the range at which a window still fits on the screen. */
A.38 StdTCP

definition module StdTCP

import StdChannels,
StdTCPChannels,
StdEventTCP,
StdStringChannels,
StdTCPRef
A.39 StdTCPChannels

definition module StdTCPChannels

    // ******************************************************************************************
    // Clean Standard Object I/O library, version 1.2
    //
    // StdTCPChannels provides instances to use TCP.
    // ******************************************************************************************

import StdTCPDef, StdChannels
from StdString import String
from StdIOCommon import OkBool
from StdIOBasic import Void, :/
from tcp/bytestreams import TCP_SCharStream, TCP_RCharStream_

    // ******************************************************************************
    // Listeners
    // ******************************************************************************

instance Receive TCP Listener_
instance closeRChannel TCP Listener_
/* Receiving on a listener will accept a TCP DuplexChannel. eom never becomes True for listeners. */

    // ******************************************************************************
    // TCP send channels
    // ******************************************************************************

instance Send TCP SChannel_

    // ******************************************************************************
    // TCP receive channels
    // ******************************************************************************

instance Receive TCP RChannel_
instance closeRChannel TCP RChannel_
instance MaxSize TCP RChannel_

    // ******************************************************************************
    // TCP char streams to receive
    // ******************************************************************************

:: TCP RCharStream ::= TCP RCharStream Char
:: TCP RCharStreams = TCP RCharStream [TCP RCharStream]
toCharStream :: !TCP RChannel -> TCP RCharStream

instance Receive TCP RCharStream_
instance closeRChannel TCP RCharStream_

    // ******************************************************************************
    // TCP char streams to send
    // ******************************************************************************

:: TCP SCharStream ::= TCP SCharStream Char
:: TCP SCharStreams = TCP SCharStream [TCP SCharStream]
toSCharStream :: !TCP SChannel -> TCP SCharStream

instance Send TCP SCharStream_

    // ******************************************************************************
    // establishing connections
    // ******************************************************************************
lookupIPAddress :: String → Maybe IPAddress 
lookupIPAddress :: String → Maybe IPAddress 
lookupIPAddress :: String → Maybe IPAddress 
lookupIPAddress :: String → Maybe IPAddress
lookupIPAddress :: String → Maybe IPAddress

connectTCP :: (!Maybe !Timeout) (!IPAddress, !Port) → Maybe TCP_DuplexChannel 
connectTCP_MyTCPListener :: Port → Maybe TCP_MyTCPListener 
tcpPossible :: !env → Bool

class SelectReceive channels where
accRChannels :: PrimitiveRChannel → (x, PrimitiveRChannel)) → channels 
getRState :: Int → channels → World 
getRState :: Int → channels → World 
getRState :: Int → channels → World

class SelectSend channels where
accSChannels :: TCP_SChannel → channels → channels 
appDisconnected :: Int → channels → World 
appDisconnected :: Int → channels → World 
appDisconnected :: Int → channels → World

selectChannel_MT :: ([!Int, !SelectResult], !r_channels, !s_channels, !World) 
selectChannel_MT :: ([!Int, !SelectResult], !r_channels, !s_channels, !World)
/\ accSChannels
  applies a function on each channel in channels and returns a list which
  contains the result for each application.
appDisconnected
  returns whether disconnected is True for the channel which is identified by
  the Int parameter.
*/
class getNrOfChannels channels :: !\channels -> (!Int, !\channels)
/* getNrOfChannels channels
  returns the number of channels in channels.
*/
instance SelectReceive TCP_RChannels, TCP_Listeners, TCP_RCharStreams, Void
instance SelectReceive (::: *x y) | SelectReceive, getNrOfChannels x
                          & SelectReceive y
instance SelectSend TCP_SChannels, TCP_SCharStreams, Void
instance SelectSend (::: *x y) | SelectSend, getNrOfChannels x
                                & SelectSend y
instance getNrOfChannels TCP_RChannels, TCP_Listeners, TCP_RCharStreams,
                          TCP_SChannels, TCP_SCharStreams, Void
instance getNrOfChannels (::: *x y) | getNrOfChannels x & getNrOfChannels y
A.40  StdTCPDef

definition module StdTCPDef

// ***************************************************************************
// Clean Standard Object I/O library, version 1.2
//
// StdTCPDef provides basic definitions for using TCP.
// ***************************************************************************

from StdMaybe import Maybe
from StdReceiverDef import Id, ReceiverFunction, ReceiverAttribute
from StdOverloaded import toString, ==
from StdChannels import DuplexChannel, ReceiveMsg, SendEvent
from tcp import TCP_SChannel, TCP_RChannel, TCP_Listener, IPAddress

:: *TCP_SChannel :~ TCP_SChannel_ ByteSeq
:: *TCP_RChannel :~ TCP_RChannel_ ByteSeq
:: *TCP_Listener :~ TCP_Listener_ (IPAddress, TCP_DuplexChannel)
:: Port :~ Int
:: *TCP_DuplexChannel :~ DuplexChannel *TCP_SChannel_ *TCP_RChannel_ ByteSeq
:: ByteSeq
// A sequence of bytes
instance toString ByteSeq
instance == ByteSeq
toByteSeq x :!x -> ByteSeq | toString x
byteSeqSize :!ByteSeq -> Int
// byteSeqSize returns the size in bytes
instance toString IPAddress
// returns ip address in dotted decimal form

// ***************************************************************************
// for event driven processing
// ***************************************************************************

// To receive byte sequences
:: *TCP_Receiver ls pst
  = TCP_Receiver
    Id TCP_RChannel
    (ReceiverFunction (ReceiveMsg ByteSeq) *(ls.pst))
    [ReceiverAttribute *(ls.pst)]
:: SendNotifier sChannel ls pst
  = SendNotifier
    sChannel
    (ReceiverFunction SendEvent *(ls.pst))
    [ReceiverAttribute *(ls.pst)]

// To accept new connections
:: *TCP_ListenerReceiver ls pst
  = TCP_ListenerReceiver
    Id TCP_Listener
    (ReceiveMsg (IPAddress,TCP_DuplexChannel)) -> *[*(ls.pst) -> *(ls.pst))
    [ReceiverAttribute *(ls.pst)]

// To receive characters
:: *TCP_CharReceiver ls pst
  = TCP_CharReceiver
    Id TCP_RChannel (Maybe NrOfIterations)
    (ReceiverFunction (ReceiveMsg Char) *(ls.pst))
[ReceiverAttribute *(ls.pst)]

/* For efficiency the receiver function of a TCP CharReceiver will be called from
a loop. Within this loop no other events can be handled. The NrOfIterations
parameter limits the maximum number of iterations.
*/
:: NrOfIterations ::= Int
:: InetLookupFunction st ::= (Maybe IPAddress) -> *(st -> st)
:: InetConnectFunction st ::= (Maybe TCP_DuplexChannel) -> *(st -> st)

// for multiplexing
// ***************************************************************

:: *TCP_RChannels :: TCP_RChannels [TCP_RChannel]
:: *TCP_SChannels :: TCP_SChannels [TCP_SChannel]
:: *TCP_Listeners :: TCP_Listeners [TCP_Listener]

:: *PrimitiveRChannel
  = TCP_RCHANNEL TCP_RChannel
  | TCP_LISTENER TCP_Listener

:: SelectResult
  = SR_Available
  | SR_EOL
  | SR_Sendable
  | SR_Disconnected
A.41 StdTime

definition module StdTime

// Clean Standard Object I/O library, version 1.2
// StdTime contains time related operations.
from StdOverloaded import <
from ostick import Tick

:: Time
  = { hours : : !Int // hours (0-23)
       , minutes : : !Int // minutes (0-59)
       , seconds : : !Int // seconds (0-59)
       }

:: Date
  = { year : : !Int // year
       , month : : !Int // month (1-12)
       , day : : !Int // day (1-31)
       , dayNr : : !Int // day of week (1-7, Sunday=1, Saturday=7)
       }

wait :: !Int .x -> .x

/* wait n x suspends the evaluation of x modally for n ticks.
   If n<0, then x is evaluated immediately. */

instance < Tick
intPlusTick :: !Int !Tick -> Tick
tickDifference :: !Tick !Tick -> Int

class TimeEnv env where
  getBlinkInterval :: !*env -> (!Int, !*env)
  getCurrentTime :: !*env -> (!Time, !*env)
  getCurrentDate :: !*env -> (!Date, !*env)
  getCurrentTick :: !*env -> (!Tick, !*env)

/* getBlinkInterval
   returns the time interval in ticks that should elapse between blinks of
   e.g. a cursor. This interval may be changed by the user while the
   interactive process is running!

getCurrentTime
   returns the current Time.

getCurrentDate
   returns the current Date.

getCurrentTick
   returns the current Tick. */

instance TimeEnv World
A.42 StdTimer

definition module StdTimer

  // ******************************************************************************
  // Clean Standard Object I/O library, version 1.2
  //
  // StdTimer specifies all timer operations.
  // ******************************************************************************

import StdTimerElementClass, StdMaybe
from StdSystem import ticksPerSecond
from iostate import PSt, IOSt

class Timers tdef where
  openTimer :: !ls !(tdef .ls (PSt .l)) !(PSt .l) -> (!ErrorReport, !PSt .l)
goalgetType :: (tdef .ls .pst) -> TimerType
  /*
  Open a new timer.
  This function has no effect in case the interactive process already contains
  a timer with the same Id. In case TimerElements are opened with duplicate Ids,
  the timer will not be opened. Negative TimerIntervals are set to zero.
  In case the timer does not have an Id, it will obtain an Id which is fresh
  with respect to the current set of timers. The Id can be reused after closing
  this timer.
  */

instance Timers (Timer t) | TimerElements t

closeTimer :: !Id !(IOSt .l) -> IOSt .l
  /* closeTimer closes the timer with the indicated Id.
  */

getTimers :: !!(IOSt .l) -> ![[(Id,TimerType)],!IOSt .l]
  /* getTimers returns the Ids and TimerTypes of all currently open timers.
  */

enableTimer :: !Id !(IOSt .l) -> IOSt .l
disableTimer :: !Id !(IOSt .l) -> IOSt .l
getTimerSelectState :: !Id !(IOSt .l) -> ![Maybe SelectState, !IOSt .l]
  /* (en/dis)ableTimer (en/dis)ables the indicated timer.
   getTimerSelectState yields the SelectState of the indicated timer. If the timer
   does not exist, then Nothing is yielded.
  */

setTimerInterval :: !Id !TimerInterval !(IOSt .l) -> IOSt .l
getTimerInterval :: !Id !(IOSt .l)
  -> ![Maybe TimerInterval, !IOSt .l]
  /* setTimerInterval
   sets the TimerInterval of the indicated timer.
   Negative TimerIntervals are set to zero.
   getTimerInterval
   yields the TimerInterval of the indicated timer.
   If the timer does not exist, then Nothing is yielded.
  */
A.43 StdTimerAttribute

definition module StdTimerAttribute

// Clean Standard Object I/O library, version 1.2
// StdTimerAttribute specifies which TimerAttributes are valid for each of the
// standard timers.
// Basic comparison operations and retrieval functions are also included.
import StdTimerDef

/* The following functions specify the valid attributes for each standard timer. */
isValidTimerAttribute : !((TimerAttribute .st)) -> Bool
/* Timer (y = valid, . = invalid)
   TimerFunction y | TimerInit y | TimerId y | TimerSelectState y | */

/* The following functions return True only iff the attribute equals the
   indicated name. */
isTimerFunction : !((TimerAttribute .st)) -> Bool
isTimerId : !((TimerAttribute .st)) -> Bool
isTimerInit : !((TimerAttribute .st)) -> Bool
isTimerSelectState : !((TimerAttribute .st)) -> Bool

/* The following functions return the attribute value if appropriate.
   THESE ARE PARTIAL FUNCTIONS! They are only defined on the corresponding
   attribute. */
getTimerFun : !((TimerAttribute .st)) -> TimerFunction .st
getTimerIdAtt : !((TimerAttribute .st)) -> Id
getTimerInitFun : !((TimerAttribute .st)) -> IdFun .st
getTimerSelectStateAtt : !((TimerAttribute .st)) -> SelectState
A.44 StdTimerDef

definition module StdTimerDef

// Clean Standard Object I/O library, version 1.2
// StdTimerDef contains the types to define the standard set of timers.
// *************************************************************************/

import StdIOCommon

:: Timer t ls pst = Timer TimerInterval (t ls pst) [TimerAttribute *(ls pst)]

:: TimerInterval
    ::= Int

:: TimerAttribute st
    // Default:
    = TimerFunction  (TimerFunction st) // _ x->x
    | TimerId        Id       // no Id
    | TimerInit      (IdFun st)  // no actions after opening timer
    | TimerSelectState SelectState // timer Able

:: TimerFunction st ::= NrOfIntervals -> st -> st

:: NrOfIntervals ::= Int

:: TimerType ::= String

:: TimerElementType ::= String
A.45 StdTimerElementClass

definition module StdTimerElementClass

// ************************************************************************
// Clean Standard Object I/O library, version 1.2
//
// StdTimerElementClass define the standard set of timer element instances.
// ************************************************************************

import StdIOCommon, StdTimerDef
from iostate import PSt, IOSt
from timerhandle import TimerElementState

class TimerElements t where
    timerElementToHandles :: !(t .ls (PSt .l)) !(PSt .l)
        -> (![TimerElementState .ls (PSt .l)], !PSt .l)
    getTimerElementType :: (t .ls .pst)
        -> TimerElementType

instance TimerElements (NewLS t) | TimerElements t
instance TimerElements (AddLS t) | TimerElements t
instance TimerElements (ListLS t) | TimerElements t
instance TimerElements NilLS
instance TimerElements (((::*)) t1 t2) | TimerElements t1
    & TimerElements t2
A.46  StdTimerReceiver

definition module StdTimerReceiver

;; ************************************************************************
;; Clean Standard Object I/O library, version 1.2
;;
;; StdTimerReceiver defines Receiver(2) timer element instances.
;; ************************************************************************

import StdReceiverDef, StdTimerElementClass

;; Receiver components for timers:
instance TimerElements (Receiver m )
instance TimerElements (Receiver2 m r)
A.47 StdWindow

definition module StdWindow

// Clean Standard Object I/O library, version 1.2
// StdWindow defines functions on windows and dialogues.

from StdFunc import St
import StdControlClass, StdWindowDef
from StdPSt import PSt, IOSt

// Functions applied to non-existent windows or unknown ids have no effect.
class Windows wdef where
    openWindow :: .ls !(wdef .ls (PSt .1)) !(PSt .1)
        -> (!ErrorReport, !PSt .1)
    getWindowType :: (wdef .ls .pst) -> WindowType

class Dialogs wdef where
    openDialog :: .ls !(wdef .ls (PSt .1)) !(PSt .1)
        -> (!ErrorReport, !PSt .1)
    openModalDialog :: .ls !(wdef .ls (PSt .1)) !(PSt .1)
        -> (!(!ErrorReport, Maybe .ls), !PSt .1)
    getDialogType :: (wdef .ls .pst) -> WindowType

    open((Window/Dialog)) opens the given window/dialog.
    If the Window/(Dialog) has no WindowIndex attribute (see StdWindowDef), then
    the new window is opened frontmost.
    If the Window/(Dialog) has a WindowIndex attribute, then the new window is
    opened behind the window indicated by the integer index:
    Index value 1 indicates the top-most window.
    Index value N indicates the bottom-most modal window, if there are N modal
    windows.
    Index value N indicates the bottom-most window, if there are N windows.
    If index<0, then the new window is added behind the bottom-most modal window
    (at index M).
    If index>N, then the new window is added behind the bottom-most window
    (at index N).
    openModalDialog always opens a window at the front-most position.
    openWindow may not be permitted to open a window depending on its
    DocumentInterface (see the comments at the ShareProcesses instances in
    module StdProcess).
    In case the window does not have an Id, it will obtain a fresh Id. The Id can
    be reused after closing this window.
    In case a window with the same Id is already open the window will not be opened.
    In case controls are opened with duplicate Ids, the window will not be opened.
    openModalDialog terminates when:
    the modal dialog has been closed (by means of closeWindow), or the process
    has been terminated (by means of closeProcess). If the ErrorReport=NoError,
    then also the final local state of the modal dialog is returned, otherwise
    Nothing.
    */

instance Windows (Window c) | Controls c
instance Dialogs (Dialog c) | Controls c

closeWindow :: !Id !(PSt .1) -> PSt .1
closeActiveWindow :: !(PSt .1) -> PSt .1

/* If the indicated window is not an inactive modal dialog, then closeWindow closes
the window/dialogue.
In case the Id of the window was generated by open(Window/Dialog), it will
become reusable for new windows/dialogues.

No window is closed in case of an unknown Id.

closeActiveWindow closes the currently active window/dialogue (see also getActiveWindow) if such a window could be found.

*/

setActiveWindow :: !Id    !(IOSt .1) -> IOSt .1
getActiveWindow ::    !(IOSt .1) -> (!Maybe Id,IIOSt .1)

/* setActiveWindow makes the indicated window the active window.
If there are modal dialogues, then the window will be placed behind the last modal dialog.
setActiveWindow has no effect in case the window is unknown or is a modal dialog.
getActiveWindow returns the Id of the window that currently has the input focus of the interactive process.
Nothing is returned if there is no such window.
*/

setActiveControl :: !Id    !(IOSt .1) -> IOSt .1
getActiveControl ::    !(IOSt .1) -> (!Maybe (!Bool,Maybe Id),IIOSt .1)

/* setActiveControl makes the indicated (PopUp/Edit/Custom/Compound)Control the active control. This succeeds only if its parent window is already active.
getActiveControl returns the Id of the (PopUp/Edit/Custom/Compound)Control that currently has the input focus.
The Boolean result is True only if such a control could be found.
Nothing is returned if the control has no Id attribute or if the Boolean result is False.
*/

stackWindow :: !Id !Id    !(IOSt .1) -> IOSt .1

/* stackWindow id1 id2 places the window with id1 behind the window with id2.
If id1 or id2 is unknown, or id1 indicates a modal window, stackWindow does nothing.
If id2 indicates a modal window, then the window with id1 is placed behind the last modal window.
*/

getWindowStack :: !(IOSt .1) -> (![(Id,WindowType)],IIOSt .1)
getWindowsStack :: !(IOSt .1) -> (![Id],IIOSt .1)
getDialogsStack :: !(IOSt .1) -> (![Id],IIOSt .1)

/* getWindowStack returns the Ids and WindowTypes of all currently open windows, in the current stacking order starting with the active window.
get/(Windows/Dialogs)Stack is equal to getWindowStack, restricted to Windows instances and Dialogs instances respectively.
*/

getDefaultHMargin :: !Bool    !(IOSt .1) -> ((Int,Int),IIOSt .1)
getDefaultVMargin :: !Bool    !(IOSt .1) -> ((Int,Int),IIOSt .1)
getDefaultItemSpace :: !Bool    !(IOSt .1) -> ((Int,Int),IIOSt .1)
getWindowHMargin :: !Id    !(IOSt .1) -> (!Maybe (Int,Int),IIOSt .1)
getWindowVMargin :: !Id    !(IOSt .1) -> (!Maybe (Int,Int),IIOSt .1)
getWindowItemSpace :: !Id    !(IOSt .1) -> (!Maybe (Int,Int),IIOSt .1)

/* getDefault/(H/V)Margin/(ItemSpace) isWindow return the default values for the horizontal and vertical window (if isWindow)/dialogue (if not isWindow) margins and item spaces.
getWindow/(H/V)Margin/(ItemSpace) return the current horizontal and vertical margins and item spaces of the indicated window. These will have the default values in case they were not specified as an attribute.
In case the window does not exist, Nothing is yielded.
*/

enableWindow :: !Id    !(IOSt .1) -> IOSt .1
disableWindow :: !Id    !(IOSt .1) -> IOSt .1
enableWindowMouse :: Id ! (IOSt .1) -> IOSt .1
disableWindowMouse :: Id ! (IOSt .1) -> IOSt .1
enableWindowKeyboard :: Id ! (IOSt .1) -> IOSt .1
disableWindowKeyboard :: Id ! (IOSt .1) -> IOSt .1

(* (en/dis)ables the indicated window.
(en/dis)ables mouse handling of the indicated window.
(en/dis)ables keyboard handling of the indicated window.
Disabling a window overrules the SelectStates of its elements, which all become Unable.
Reenabling the window reestablishes the SelectStates of its elements.
The functions have no effect in case of invalid Ids or Dialogs instances.
The latter four functions also have no effect in case the Window does not have the indicated attribute.
*)

getWindowSelectState :: Id ! (IOSt .1) -> (!Maybe SelectState, IOSt .1)
getWindowHouseSelectState :: Id ! (IOSt .1) -> (!Maybe SelectState, IOSt .1)
getWindowKeyboardSelectState :: Id ! (IOSt .1) -> (!Maybe SelectState, IOSt .1)

(* getWindowSelectState yields the current SelectState of the indicated window.
getWindowHouse/KeyboardSelectState yields the current SelectState of the mouse/keyboard of the indicated window.
The functions return Nothing in case of invalid Ids or Dialogs instances or if the Window does not have the indicated attribute.
*)

getWindowHouseStateFilt er :: Id ! MouseStateFilter ! (IOSt .1)
getWindowKeyboardStateFilt er :: Id ! KeyboardStateFilter ! (IOSt .1)

(* getWindowHouse/KeyboardStateFilter yields the current House/KeyboardStateFilter of the indicated window. Nothing is yielded in case the window does not exist or has no Window/House/Keyboard attribute.
setWindowHouse/KeyboardStateFilter replaces the current House/KeyboardStateFilter of the indicated window. If the indicated window does not exist the function has no effect.
*)

appWindowPicture :: !Id !(IdFun *Picture) !(IOSt .1) -> IOSt .1
accWindowPicture :: !Id !(St *Picture .x) !(IOSt .1) -> (!Maybe .x, IOSt .1)

(* (app/acc)WindowPicture applies the given drawing function to the Picture of the indicated window (behind all controls).
Both functions have no effect in case the window is unknown or is a Dialog.
In that case, accWindowPicture also returns Nothing.
*)

updateWindow :: Id !(Maybe ViewFrame) !(IOSt .1) -> IOSt .1

(* updateWindow applies the WindowLook attribute function of the indicated window.
The Look attribute function is applied to the following arguments:
The current SelectState of the window, and
the UpdateState argument
{oldFrame=viewframe,newFrame=viewframe,updateArea=[frame]}
where viewframe is the current ViewFrame of the window;
and frame depends on the optional ViewFrame argument:
in case of (Just rectangle):

A.47. STDWINDOW 273
the intersection of viewframe and rectangle.
in case of Nothing: viewframe.
updateWindow has no effect in case of unknown windows, or if the indicated
window is a Dialog, or the optional viewframe argument is an empty rectangle.

*/

setWindowLook :: !Id !Bool !((Bool, !Look) !(IOSt .1) -> IOSt .1
getWindowLook :: !Id 

-> ((Maybe (Bool, Look), IOSt .1)

*/ setWindowLook sets the (render/look) of the indicated window.
The window is redrawn only if the Boolean argument is True.
setWindowLook has no effect in case the window does not exist, or is a Dialog.
getWindowLook returns the (Just (render/look)) of the indicated window.
In case the window does not exist, or is a Dialog, or has no WindowLook
attribute, the result is Nothing.

*/

setWindowPos :: !Id !ItemPos !(IOSt .1) -> IOSt .1
getWindowPos :: !Id 

-> ((Maybe Vector/2!, IOSt .1)

*/ setWindowPos places the window at the indicated position.
If the ItemPos argument refers to the Id of an unknown window (in case of
LeftOf/RightTo/Above/Below), setWindowPos has no effect.
If the ItemPos argument is one of (LeftOf/RightTo/Above/BelowPrev), then the
previous window is the window that is before the window in the current
stacking order.
If the window is frontmost, setWindowPos has no effect. setWindowPos also
has no effect if the window would be moved outside the screen, or if the Id
is unknown or refers to a modal Dialog.
getWindowPos returns the current item offset position of the indicated window.
The corresponding ItemPos is (LeftTop, OffsetVector offset). Nothing is
returned in case the window does not exist.

*/

moveWindowViewFrame :: !Id Vector/2!!(IOSt .1) -> IOSt .1
getWindowViewFrame :: !Id 

-> ((ViewFrame, IOSt .1)

*/ moveWindowViewFrame moves the orientation of the view frame of the indicated
window over the given vector, and updates the window if necessary. The view
frame is not moved outside the ViewDomain of the window.
In case of unknown Id, or of Dialogs, moveWindowViewFrame has no effect.

*/

ggetWindowViewFrame :: !Id !IOSt .1) -> (!ViewFrame, !IOSt .1)

*/ getWindowsViewFrame returns the current view frame of the window in terms of the
ViewDomain. Note that in case of a Dialog, getWindowsViewFrame returns
(zero, size).
In case of unknown windows, the ViewFrame result is zero.

*/

setWindowViewSize :: !Id !Size !(IOSt .1) -> IOSt .1
getWindowViewSize :: !Id 

-> ((Size, IOSt .1)

*/ setWindowViewSize
sets the size of the view frame of the indicated window as given, and
updates the window if necessary. The size is fit between the minimum size
and the screen dimensions.
In case of unknown Ids, or of Dialogs, setWindowViewSize has no effect.
setWindowViewSize yields the current size of the view frame of the indicated
window. If the window does not exist, zero is returned.

*/

setWindowOuterSize :: !Id !Size !(IOSt .1) -> IOSt .1
getWindowOuterSize :: !Id 

-> ((Size, IOSt .1)

*/ setWindowOuterSize
sets the size of the outer frame of the indicated window as given, and
updates the window if necessary. The size is fit between the minimum size
and the screen dimensions.
In case of unknown IDs, or of Dialogs, setWindowOuterSize has no effect.
getWindowOuterSize yields the current size of the outer frame of the indicated
window. If the window does not exist, zero is returned.

/*
setWindowViewDomain :: !Id ViewDomain !(IOSt .1) -> IOSt .1
getWindowViewDomain :: !Id !(IOSt .1)
  -> (!Maybe ViewDomain, !IOSt .1)
/* setWindowViewDomain
sets the view domain of the indicated window as given. The window view frame
is moved such that a maximum portion of the view domain is visible. The
window is not resized.
In case of unknown IDs, or of Dialogs, setWindowViewDomain has no effect.
getWindowViewDomain
returns the current ViewDomain of the indicated window.
Nothing is returned in case the window does not exist or is a Dialog.

/*/  
setWindowScrollFunction :: !Id Direction ScrollFunction !(IOSt .1) -> IOSt .1
getWindowScrollFunction :: !Id Direction !(IOSt .1)
  -> (!Maybe ScrollFunction, !IOSt .1)
/* setWindowScrollFunction
changes the current scroll function of the indicated Window and direction
only if the indicated window already had a scroll bar in that direction.
In all other cases setWindowScrollFunction has no effect.
getWindowScrollFunction
returns the current scroll function in the argument direction if the
indicated Window had one.
In all other cases Nothing is returned.

/*/  
setWindowTitle :: !Id Title !(IOSt .1) -> IOSt .1
setTitle :: !Id Id !(IOSt .1) -> IOSt .1
setTitle :: !Id CursorShape !(IOSt .1) -> IOSt .1
getWindowTitle :: !Id !(IOSt .1) -> (!Maybe Title, !IOSt .1)
getTitle :: !Id !(IOSt .1) -> (!Maybe Title, !IOSt .1)
getTitle :: !Id !(IOSt .1) -> (!Maybe Title, !IOSt .1)
getTitle :: !Id !(IOSt .1) -> (!Maybe Title, !IOSt .1)
/* setWindowTitle/Ok/Cancel/Cursor) set the indicated window attributes.
In case of unknown IDs, these functions have no effect.
getWindowTitle/Ok/Cancel/Cursor) get the indicated window attributes.
In case of unknown IDs, the result is Nothing.

/*/
A.48 StdWindowAttribute

definition module StdWindowAttribute

// ******************************************************
// Clean Standard Object I/O library, version 1.2
// StdWindowAttribute specifies which WindowAttributes are valid for Windows
// and Dialogs.
// Basic comparison operations and retrieval functions are also included.
// ******************************************************

import StdWindowDef

/* The following functions specify the valid attributes for each standard window. */
isValidWindowAttribute :: ! (WindowAttribute .st) -> Bool
  Window Activate y | Window Init y | Window Pen y |
  Window Cancel . | Window Init Active y | Window Pos y |
  Window Close y | Window Item Space y | Window Select State y |
  Window Cursor y | Window Keyboard y | Window View Domain y |
  Window Deactivate y | Window Ok . | Window Scroll y |
  Window Id y | Window Origin y |
  Window Index y | Window Outer Size y |
/*

isValidDialogAttribute :: ! (WindowAttribute .st) -> Bool
  Dialog Activate y | Window Init y | Window Pen . |
  Window Cancel y | Window Init Active y | Window Pos y |
  Window Close y | Window Item Space y | Window Select State . |
  Window Cursor . | Window Keyboard . | Window View Domain . |
  Window Deactivate y | Window Ok . | Window Scroll y |
  Window H Margin y | Window House . | Window W Margin y |
  Window H Scroll . | Window Ok y | Window Scroll . |
  Window Id y | Window Origin . |
  Window Index y | Window Outer Size y |
/*

/* The following functions return True only iff the attribute equals the
indicated name. */
isWindowActivate :: ! (WindowAttribute .st) -> Bool
isWindowCancel :: ! (WindowAttribute .st) -> Bool
isWindowClose :: ! (WindowAttribute .st) -> Bool
isWindowCursor :: ! (WindowAttribute .st) -> Bool
isWindowDeactivate :: ! (WindowAttribute .st) -> Bool
isWindowHMargin :: ! (WindowAttribute .st) -> Bool
isWindowHScroll :: ! (WindowAttribute .st) -> Bool
isWindowId :: ! (WindowAttribute .st) -> Bool
isWindowIndex :: ! (WindowAttribute .st) -> Bool
isWindowInit :: ! (WindowAttribute .st) -> Bool
isWindowInitActive :: ! (WindowAttribute .st) -> Bool
isWindowItemSpace :: ! (WindowAttribute .st) -> Bool
isWindowKeyboard :: ! (WindowAttribute .st) -> Bool
isWindowLook :: ! (WindowAttribute .st) -> Bool
isWindowMouse :: ! (WindowAttribute .st) -> Bool
isWindowOk :: ! (WindowAttribute .st) -> Bool
isWindowOrigin :: ! (WindowAttribute .st) -> Bool
isWindowOuterSize :: !WindowAttribute .st) -> Bool
isWindowPen :: !WindowAttribute .st) -> Bool
isWindowPos :: !WindowAttribute .st) -> Bool
isWindowSelectState :: !WindowAttribute .st) -> Bool
isWindowViewDomain :: !WindowAttribute .st) -> Bool
isWindowSize :: !WindowAttribute .st) -> Bool
isWindowMargin :: !WindowAttribute .st) -> Bool
isWindowState :: !WindowAttribute .st) -> Bool

/* The following functions return the attribute value if appropriate.
   THESE ARE PARTIAL FUNCTIONS! They are only defined on the corresponding
   attribute. */
getWindowActivateFun :: !WindowAttribute .st) -> IdFun .st
getWindowCancelAtt :: !WindowAttribute .st) -> Id
getWindowCloseFun :: !WindowAttribute .st) -> IdFun .st
getWindowCursorAtt :: !WindowAttribute .st) -> CursorShape
getWindowDeactivateFun :: !WindowAttribute .st) -> IdFun .st
getWindowHMarginAtt :: !WindowAttribute .st) -> (Int,Int)
getWindowHScrollFun :: !WindowAttribute .st) -> ScrollFunction
getWindowIdAtt :: !WindowAttribute .st) -> Id
getWindowIndexAtt :: !WindowAttribute .st) -> Int
getWindowInitFun :: !WindowAttribute .st) -> IdFun .st
getWindowInitActiveAtt :: !WindowAttribute .st) -> Id
getWindowItemSpaceAtt :: !WindowAttribute .st) -> (Int,Int)
getWindowKeyboardAtt :: !WindowAttribute .st) -> (KeyboardStateFilter
    , SelectState
    , KeyboardFunction .st
  )
getWindowLookAtt :: !WindowAttribute .st) -> (Bool,Look)
getWindowMouseAtt :: !WindowAttribute .st) -> (MouseStateFilter
    , SelectState
    , MouseFunction .st
  )
getWindowOkAtt :: !WindowAttribute .st) -> Id
getWindowOriginAtt :: !WindowAttribute .st) -> Point2
getWindowOuterSizeAtt :: !WindowAttribute .st) -> Size
getWindowPenAtt :: !WindowAttribute .st) -> [PenAttribute]
getWindowPosAtt :: !WindowAttribute .st) -> ItemPos
getWindowSelectStateAtt :: !WindowAttribute .st) -> SelectState
getWindowViewDomainAtt :: !WindowAttribute .st) -> ViewDomain
getWindowViewSizeAtt :: !WindowAttribute .st) -> Size
getWindowMarginAtt :: !WindowAttribute .st) -> (Int,Int)
getWindowScrollFun :: !WindowAttribute .st) -> ScrollFunction
A.49 StdWindowDef

```haskell
import StdIOCommon, StdPictureDef

:: Dialog c ls pst = Dialog Title (c ls pst) [WindowAttribute *(ls,pst)]
:: Window c ls pst = Window Title (c ls pst) [WindowAttribute *(ls,pst)]

:: WindowAttribute st // Default:
// Attributes for Windows and Dialogs:
- WindowActivate (IdFun st) // id
- WindowClose (IdFun st) // user can't close window
- WindowDeactivate (IdFun st) // id
- WindowHMargin Int Int // system dependent
- WindowId Id // system defined id
- WindowIndex Int // open front-most
- WindowInit (IdFun st) // no actions after opening window
- WindowInitActive Id // system dependent
- WindowItemSpace Int Int // system dependent
- WindowOuterSize Size // screen size
- WindowPos ItemPos // system dependent
- WindowViewSize Size // screen size
- WindowVMargin Int Int // system dependent
// Attributes for Dialog only:
- WindowCancel Id // no cancel (Custom)ButtonControl
- WindowOk Id // no default (Custom)ButtonControl
// Attributes for Windows only:
- WindowCursor CursorShape // no change of cursor
- WindowHScroll ScrollFunction // no horizontal scrolling
- WindowKeyboard KeyboardStateFilter SelectState (KeyboardFunction st)
- WindowLook Bool Look // show system dependent background
- WindowMouse MouseStateFilter SelectState (MouseFunction st)
- WindowOrigin Point2 // left top of picture domain
- WindowPen [PenAttribute] // default pen attributes
- WindowSelectState SelectState // Able
- WindowViewDomain ViewDomain // (zero, max range)
- WindowVScroll ScrollFunction // no vertical scrolling

:: WindowType :: String
```
getChar, 165, 166, 245
restorePos, 165, 166, 245
savePos, 165, 166, 245
charStreamToFile, 167, 245
chat, see TCP
CheckControl, see control
CheckControlItem, 207
circle, see oval
class
accScreenPicture, see accScreen-Picture
CharStreams, see CharStreams
Clipboard, see Clipboard
Controls, see Controls
controlSize, see controlSize
Dialogs, see Dialogs
Drawables, see Drawables
FileSelectEnv, see FileSelectEnv
Fillables, see Fillables
Hilites, see Hilites
Ids, see Ids
MenuElements, see MenuElements
Menus, see Menus
movePenPos, see movePenPos
Processes, see Processes
Receivers, see Receivers
shareProcesses, see shareProcesses
TimeEnv, see TimeEnv
TimerElements, see TimerElements
Timers, see Timers
toRegion, see Region
Windows, see Windows
ClearKey, see SpecialKey
clearMaxSize, 192
client, see TCP
Clipboard, 153, 195
fromClipboard, 153, 195
toClipboard, 153, 195
clipboard, 110, 153
clipboardHasChanged, 154, 195
ClipboardItem, 153, 195
cropping, 46
—region, 47
close command, 111
closeActiveWindow, 54, 69, 112, 271
closeAllControls, 196
closeChannel, 192
closeChannel_MT, 192
closeControls, 196
closeMenu, 224
closeMenuElements, 224
closeMenuItemElements, 224
closePopupControlItems, 196
closeProcess, 17, 139, 147, 247
closeRadioMenuIndexElements, 224
closeChannel, 192
closeReceiver, 252
closeSendNotifier, 210
closeSubMenuItemElements, 224
closeTimer, 266
closeWindow, 54, 271
Colour, 31, 240
Black, 31, 240
Blue, 31, 240
Cyan, 31, 240
DarkGrey, 31, 240
Green, 31, 240
Grey, 31, 240
LightGrey, 31, 240
Magenta, 31, 240
Red, 31, 240
RGB, 240
RGBColour, 31
White, 31, 240
Yellow, 31, 240
Columns, see RowsOrColumns
command, see menu
CommandOnly, 216
CompoundOnly, 216
CompoundOnly, 216
connection
—establishment, 173
tearing down, 179
connectTCP, see TCP
connectTCPasync, 210
connectTCP_MT, 260
context switch, 127
control, 49, 73
—attribute, 74
—attribute, 207
ControlActivate, 74, 207
ControlDeactivate, 74, 207
ControlFunction, 74, 133, 207
ControlHide, 75, 207
ControlHMargin, 85, 86, 207
ControlHScroll, 86, 207
ControlId, 75, 207
ControlItemSpace, 85, 86, 207
ControlKeyboard, 75, 86, 207
ControlLook, 207
ControlMinimumSize, 75, 85, 86, 207
ControlModsFunction, 74, 207
ControlMouse, 75, 86, 207
ControlOrigin, 86, 207
ControlOuterSize, 85, 86, 207
ControlPen, 75, 207
ControlPos, 75, 90, 207
ControlResize, 75, 85, 86, 94, 207
ControlSelectState, 75, 207
ControlTip, 75, 207
ControlViewDomain, 207
ControlViewSize, 85, 86, 207
ControlVMargin, 85, 86, 207
ControlVScroll, 86, 207
ControlWidth, 75, 207
button—, 68, 69, 76, 133, 154, 155, 206, 207
check—, 77, 206, 207
compound—, 13, 85, 95, 96, 122, 206, 207
compound frame, 86
custom—, 78, 95, 96, 206, 207
custom button—, 206, 207
customised—, 73
edit—, 79, 154, 155, 206, 207
hierarchical—, 74
—layout, see layout
layout—, 69, 84, 132, 206, 207
platform standard—, 73
pop up—, 80
popup—, 206, 207
previous—, 94
radio—, 81, 206, 207
receiver, 126
resize, 94
slider
region, 83
slider—, 82, 206, 207
text—, 68, 84, 121, 206, 207
ControlActivateFun, 202
ControlDeactivateFun, 202
ControlFun, 202
ControlFunction, see control
ControlHide, see control
ControlHMargin, see control
ControlHMarginAtt, 202
ControlHScroll, see control
ControlHScrollFun, 202
ControlId, see control
ControlIdAtt, 202
ControlItemSpace, see control
ControlItemSpaceAtt, 202
ControlKeyboard, see control
ControlKeyboardAtt, 202
ControlLook, see control
ControlLookAtt, 202
ControlMinimumSize, see control
ControlMinimumSizeAtt, 202
ControlModsFun, 202
ControlModsFunction, see control
ControlMouse, see control
ControlMouseAtt, 202
ControlOnly, 216
ControlOrigin, see control
ControlOriginAtt, 202
ControlOuterSizeAtt, 202
ControlPenAtt, 202
ControlPos, see control
ControlPosAtt, 202
ControlResize, see control
ControlResizeFun, 202
ControlResizeFunction, 207
Controls, 12, 119, 206
controlToHandles, 206
getControlType, 206
ControlSelectState, see control
ControlSelectStateAtt, 202
ControlSize, see control
controlSize, 130, 196
ControlTipAtt, 202
controlToHandles, see Controls
ControlType, 207
ControlViewDomain, see control
ControlViewDomainAtt, 202
ControlViewSizeAtt, 202
ControlVMargin, see control
ControlVMarginAtt, 202
ControlVScroll, see control
ControlVScrollFun, 202
ControlWidth, 207
ContentWidth, 207
PixelWidth, 207
TextWidth, 207
ControlWidthAtt, 202
copy command, 110
CrossCursor, see CursorShape
CursorShape, 216
ArrowCursor, 216
BusyCursor, 216
CrossCursor, 216
FatCrossCursor, 216
HiddenCursor, 216
IBeamCursor, 216
StandardCursor, 216
Curve, 39, 240
CustomButtonControl, see control
CustomControl, see control
cut command, 110
Cyan, see Colour
DarkGrey, see Colour
Date, 265
defaultPrintSetup, see printing
defaultPrintSetup, 242
DeleteKey, see SpecialKey
descent, see font
Dialog, 49, 278
Dialogs, 14, 54, 66, 69, 271
getDialogType, 271
openDialog, 54, 67, 271
openModalDialog, 66, 271
dialogue, 13, 49, 121, 132, 154
active—, 51, 61
—frame, 50, 60
modal—, 49, 61, 62, 66
modeless—, 51
direction, 82, 216
Horizontal, 82, 216
Vertical, 82, 216
dirseparator, 235
disableControl, 196
disableControls, 196
disableMenuElements, 109, 230
disableMenus, 109, 224
disableMenuSystem, 109, 224
disableReceivers, 252
disableTimer, 148, 266
disableWindow, 271
disableWindowKeyboard, 271
disableWindowMouse, 271
Disconnected, 192
disconnected, 192
DNS, see TCP
document interface, 139
multiple—, 108, 111, 140
no—, 139
single—, 140
document interface, 216
MDI, 216
NDI, 216
SDI, 216
donkey, see SpecialKey
dpi, 160
draw, see Drawables
Drawables, 31, 36, 191, 234
draw, 31, 234
drawAt, 31, 162, 163, 234
undraw, 31, 234
undrawAt, 31, 234
drawAt, see Drawables
drawing, 22, 29
api, 22
bitmap, 44
Drawables(Bitmap), 45
box
Drawables(Box), 42
Fillables(Box), 42
circle, see oval, 278
clipping, see clipping
coordinate system, 29
curve
Drawables(Curve), 39
Fillables(Curve), 40
—environment, 29
hiliting, see Hilites
line, 36
drawLine, 36
drawLineTo, 36
Drawables(Vector2), 36
oval
Drawables(Oval), 38
Fillables(Oval), 38
Picture, 29
pixel, 30
point, 36
drawPointAt, 36
polygon
Drawables(Polygon), 43
Fillables(Polygon), 43
rectangle
Drawables(Rectangle), 41
Fillables(Rectangle), 41
text
Drawables(Char), 37
Drawables(String), 37
XOR mode, 45, 46, 161
drawLine, 234
drawLineTo, 234
drawPoint, 234
drawPointAt, 234
DuplexChannel, 192, 263
EditControl, see control
enableControl, 196
enableControls, 196
enabled, 216
enableMenuElements, 109, 230
enableMenus, 109, 224
enableMenuSystem, 109, 224
enableReceivers, 252
enableTimer, 148, 266
enableWindow, 271
enableWindowKeyboard, 271
enableWindowMouse, 271
endkey, see SpecialKey
EnterKey, see SpecialKey
EOM, 192
eom, 192
eos, see CharStreams
ErrorIdsInUse, see ErrorReport
ErrorReport, 26, 216
INDEX

ErrorIdsInUse, 26, 216
ErrorNotifierOpen, 216
ErrorUnknownObject, 216
ErrorViolateDI, 216
NoError, 26, 216
OtherError, 216
ErrorUnknownObject, see ErrorReport
ErrorViolateDI, see ErrorReport
EscapeKey, see SpecialKey
event
  abstract—, 125
  keyboard—, 62
  message—, see message
  mouse—, 65
  timer—, 117
exit command, 111
F10Key, see SpecialKey
F11Key, see SpecialKey
F12Key, see SpecialKey
F13Key, see SpecialKey
F14Key, see SpecialKey
F15Key, see SpecialKey
F1Key, see SpecialKey
F2Key, see SpecialKey
F3Key, see SpecialKey
F4Key, see SpecialKey
F5Key, see SpecialKey
F6Key, see SpecialKey
F7Key, see SpecialKey
F8Key, see SpecialKey
F9Key, see SpecialKey
FatCrossCursor, see CursorShape
FileCharStream, 167, 245
Files, 159
FileSelectEnv, 58, 111, 211
  selectDirectory, 58, 211
  selectInputFile, 58, 211
  selectOutputFile, 58, 66, 211
fileToCharStream, 167, 245
fill, see Fillables
Fillables, 31, 234
  fill, 31, 234
  fillAt, 31, 234
  unfill, 31, 234
  unfillAt, 31, 234
fillAt, see Fillables
Fix, see ItemPos
flushBuffer_MT, 192
flushBuffer_NB, 192
Font, 32
font, 32, 162
  —metrics, 33
  ascent, 33
baseline, 33
descent, 33
FontMetrics, 33
leading, 33
max. width, 33
non-proportional—, 33
proportional—, 33
TrueType—, 162
FontDef, 32, 166, 240
fontLineHeight, 163, 240
FontMetrics, 240
FontName, 240
FontSize, 240
FontStyle, 240
footer, see printing
freadPrintSetup, 242
fromClipboard, see Clipboard
fromJust, 223
fwritePrintSetup, 242
getActiveControl, 271
getActiveWindow, 61, 271
getBitmapSize, 44, 59, 191
getBlinkInterval, see TimeEnv
getChar, see CharStreams
goCheckControlItem, 196
goCheckControlItem, 196
getCheckControlSelection, 196
goCheckControlSelections, 196
goClipboard, 153, 195
goCompoundMenuElementTypes, 230
goCompoundTypes, 196
goControlItemSpace, 196
goControlItemSpaces, 196
goControlLayout, 196
goControlLayouts, 196
goControlLook, 196
goControlLooks, 196
goControlMargin, 196
goControlMargins, 196
goControlMinimumSize, 196
goControlMinimumSizes, 196
goControlNrLine, 196
goControlNrLines, 196
goControlOuterSize, 196
goControlOuterSizes, 196
goControlResize, 196
goControlResizes, 196
goControlScrollFunction, 196
goControlScrollFunctions, 196
goControlSelectState, 196
goControlSelectStates, 196
goControlShowState, 196
goControlShowStates, 196
getControlText, 129, 196
getControlTexts, 196
getControlType, see Controls
getControlTypes, 196
getControlViewDomain, 196
getControlViewDomains, 196
getControlViewFrame, 196
getControlViewFrames, 196
getControlViewSize, 196
getControlViewSizes, 196
gcurrentDate, see TimeEnv
gcurrentTime, see TimeEnv
gdefaultHMargin, 271
gdefaultItemSpace, 271
gdefaultVMargin, 271
gDialogsStack, 271
gDialogType, see Dialogs
gDocumentInterface, 250
getFontCharWidth, 34, 234
getFontCharWidths, 34, 234
gFontDef, 234
getFontMetrics, 33, 234
getFontNames, 32, 234
getFontSizes, 32, 234
gFontStringLength, 34, 234
getFontStringLengths, 34, 234
gFontStyles, 32, 234
gKeyboardStateKey, 216
gKeyboardStateKeyType, 216
gMaxSize, 192
gMenu, 230
gMenuElementMarkState, 230
gMenuElementMarkStates, 230
gMenuElementSelectState, 230
gMenuElementSelectStates, 230
gMenuElementShortKey, 230
gMenuElementShortKeys, 230
gMenuElementType, 230
gMenuElementTypes, 230
gMenuElementType, see MenuElements
gMenuElementTypes, 230
gMenuFun, 227
gMenuIdAtt, 227
gMenuIndexAtt, 227
gMenuInitFun, 227
gMenuMarkStateAtt, 227
gMenuModsFun, 227
gMenuPos, 224
gMenus, 224
gMenuSelectState, 224
gMenuSelectStateAtt, 227
gMenuStateKeyAtt, 227
gMenuTitle, 224
gMenuType, see Menus
gMouseStateButtonState, 216
gMouseStateModifiers, 216
gMouseStatePos, 216
getNrofChannels, 260
gPageSize, see printing
gPageSize, 242
gParentId, 212
gParentMenu, 230
gParentWindow, 129, 196
gPenAttributes, 234
gPenBack, 234
gPenColour, 234
gPenFont, 234
gPenFontCharWidth, 34, 234
gPenFontCharWidths, 34, 234
gPenFontMetrics, 33, 163, 234
gPenFontStringLength, 34, 63, 163, 234
gPenFontStringLengths, 34, 234
gPenPos, 234
gPenSize, 234
gPopUpControlItem, 196
gPopUpControlItems, 196
gPopUpControlSelect, 196
gPopUpControlSelections, 196
gProcessActivateFun, 248
gProcessCloseFun, 248
gProcessDeactivateFun, 248
gProcessOpenFilesFun, 248
gProcessToolbarAtt, 248
gProcessWindowPos, 247
gProcessWindowPosAtt, 248
gProcessWindowResizeFun, 248
gProcessWindowSize, 247
gProcessWindowSizeAtt, 248
gRadioControlItem, 196
gRadioControlItems, 196
gRadioControlSelect, 196
gRadioControlSelections, 196
gReceiverConnected Receivers, 254
gReceiverInitFun, 254
gReceivers, 252
gReceiverSelectState, 252
gReceiverSelectStateAtt, 254
gReceiverType, see Receivers
gRegionBound, 234
gResolution, 234
gSelectedRadioMenuItem, 230
gSelectedRadioMenuItems, 230
gSliderDirection, 196
gSliderDirections, 196
gSliderState, 196
gSliderStates, 196
getTimerElementType, see TimerElements
getTimerFun, 267
getTimerIdAtt, 267
getTimerInitFun, 267
getTimerInterval, 266
getTimers, 266
getTimerSelectState, 266
getTimerSelectStateAtt, 267
getTimerType, see Timers
getWindow, 196
getWindowActivateFun, 276
getWindowCancel, 271
getWindowCancelAtt, 276
getWindowCloseFun, 276
getWindowCursor, 271
getWindowCursorAtt, 276
getWindowDeactivateFun, 276
getWindowKeyboardSelectState, 276
getWindowKeyboardStateFilter, 271
getWindowLook, 57, 271
getWindowLookAtt, 276
getWindowInitActiveAtt, 276
getWindowInitFun, 276
getWindowItemSpace, 271
getWindowItemSpaceAtt, 276
getWindowKeyboardAtt, 276
getWindowKeyboardSelectState, 271
getWindowKeyboardStateFilter, 271
getWindowPos, 271
getWindowPosAtt, 276
getWindowScrollFunction, 271
getWindowSelectState, 271
getWindowSelectStateAtt, 276
getWindowStack, 112, 271
getWindowStack, 61, 271
getWindowTitle, 271
getWindowType, see Windows
getWindowViewDomain, 271
getWindowViewDomainAtt, 276
getWindowViewFrame, 271
getWindowViewSize, 271
getWindowViewSizeAtt, 276
getWindowWMargin, 271
getWindowWMarginAtt, 276
getWindowWScrollFun, 276
Green, see Colour
Grey, see Colour
header, see printing
HelpKey, see SpecialKey
HiddenCursor, see CursorShape
hideControl, 196
hideControls, 196
hideProcess, 247
hilite, see Hilites
hiliteAt, see Hilites
Hilites
   Hilites, 161
   Hilites, 32, 46, 234
   Box, 46
   hilite, 32, 234
   hiliteAt, 32, 234
   Rectangle, 46
hinch, 258
hmm, 258
homepath, 258
Horizontal, see Direction
I/O state, 15
IBeamCursor, see CursorShape
Id, see identification
identification, 25
—assignment rule, 25
—attribute, 25
   Id, 25, 212
   R2Id, 25, 126, 135, 212
   RId, 25, 125, 143, 212
IdFun, 214
Ids, 25, 212
   openId, 25, 212
   openIds, 25, 212
   openR2Id, 25, 212
   openR2Ids, 25, 212
   openRId, 25, 212
   openRIds, 25, 212
Index, 80, 81, 105, 214
InetConnectFunction, 263
InetLookupFunction, 263
initialisation, 17
input
—focus, 51, 62, 79, 141
   keyboard—, 49, 62, 79, 95
   mouse—, 49, 62, 79, 99
interactive
—object, 11
process, see process
internet, 172
intPlusTick, 205
IOSt, 15, 250
empty, 17
IP address, see TCP
IsCharKey, see Key
isControlActivate, 202
isControlDeactivate, 202
isControlFunction, 202
isControlHide, 202
isControlHMargin, 202
isControlHScroll, 202
isControlId, 202
isControlItemSpace, 202
isControlKeyboard, 202
isControlLook, 202
isControlMinimumSize, 202
isControlModsFunction, 202
isControlMouse, 202
isControlOrigin, 202
isControlOuterSize, 202
isControlPen, 202
isControlPos, 202
isControlResize, 202
isControlSelectState, 202
isControlTip, 202
isControlViewDomain, 202
isControlViewSize, 202
isControlVMargin, 202
isControlVScroll, 202
isEmptyRegion, 202
isJust, 234
isMenuFunction, 227
isMenuId, 227
isMenuIndex, 227
isMenuItemState, 227
isMenuMarkState, 227
isMenuModeFunction, 227
isMenuSelectState, 227
isMenuItemState, 227
isNothing, 223
isProcessActivate, 248
isProcessClose, 248
isProcessDeactivate, 248
isProcessKindAttribute, 248
isProcessNoWindowMenu, 248
isProcessOpenFiles, 248
isProcessToolbar, 248
isProcessWindowPos, 248
isProcessWindowResize, 248
isProcessWindowSize, 248
isReceiverConnectedReceivers, 254
isReceiverInit, 254
isReceiverSelectState, 254
IsRepeatKey, 216
IsSpecialKey, see Key
isTimerFunction, 267
isTimerId, 267
isTimerSelectState, 267
isValidButtonControlAttribute, 202
isValidCheckControlAttribute, 202
isValidCompoundControlAttribute, 202
isValidCustomButtonControlAttribute, 202
isValidCustomControlAttribute, 202
isValidDialogAttribute, 202
isValidEditControlAttribute, 202
isValidLayoutControlAttribute, 202
isValidMenuAttribute, 227
isValidMenuItemAttribute, 227
isValidPopUpControlAttribute, 202
isValidRadioControlAttribute, 202
isValidRadioMenuAttribute, 227
isValidReceiver2Attribute, 254
isValidReceiverAttribute, 254
isValidSliderControlAttribute, 202
isValidSubMenuAttribute, 227
isValidTextControlAttribute, 202
isValidTimerAttribute, 267
isValidWindowAttribute, 276
isWindowActivate, 276
isWindowCancel, 276
isWindowClose, 276
isWindowCursor, 276
isWindowDeactivate, 276
isWindowHMargin, 276
isWindowHScroll, 276
isWindowId, 276
isWindowIndex, 276
isWindowInit, 276
isWindowInitActive, 276
isWindowItemSpace, 276
isWindowKeyboard, 276
isWindowLook, 276
isWindowMouse, 276
isWindowOk, 276
isWindowOrigin, 276
isWindowOuterSize, 276
isWindowPen, 276
isWindowPos, 276
isWindowSelectState, 276
isWindowViewDomain, 276
isWindowViewSize, 276
INDEX

isWindowVMargin, 276
isWindowVScroll, 276
ItalicsStyle, 35, 240
item space, 68
ItemLoc, 90, 216
ItemOffset, 90, 216
NoOffset, 216
OffsetFun, 216
OffsetVector, 216
ItemPos, 90, 216
Above, 91, 216
AbovePrev, 91, 94, 216
Below, 91, 216
BelowPrev, 91, 94, 216
Center, 90, 93, 216
Fix, 90, 92, 216
Left, 90, 93, 216
LeftBottom, 90, 93, 216
LeftOf, 91, 216
LeftOff, 91, 94, 216
LeftTop, 90, 93, 216
Right, 90, 93, 216
RightBottom, 90, 93, 216
RightTo, 91, 216
RightTop, 90, 93, 216
RightToPrev, 91, 94, 216
JobInfo, see printing
JobInfo, 242
Just, see Maybe

isCharKey, 216
IsSpecialKey, 216
KeyboardFunction, 62, 96, 216
KeyboardState, 62, 216
CharKey, 62, 216
KeyLost, 216
SpecialKey, 62, 216
KeyboardStateFilter, 62, 216
KeyDown, see KeyState
KeyState, 62, 216
KeyDown, 216
KeyUp, 216
KeyUp, see KeyState

layout
  boundary aligned, 90, 92
control—, 89
default—, 91
fixed position, 90, 92
line aligned, 90, 93
offset, 93
relative—, 91, 94
  —root, 91
  —scope, 84
  —tree, 91
window—
cascade, 109
tile, 109
leading, see font
Left, see ItemPos
LeftBottom, see ItemPos
LeftJustify, see WrapMode
LeftKey, see SpecialKey
LeftOff, see ItemPos
LeftOffPrev, see ItemPos
LeftTop, see ItemPos
life-cycle, 14
  Picture, 29
LightGrey, see Colour
Line2, 240
ListCS, 214
listening, see TCP
ListLS, 68, 69, 87, 107, 118, 122, 206, 213
Look, 55, 78, 79, 96, 164, 216
lookupIPAddress, see TCP
lookupIPAddress, 200
lookupIPAddressAsync, 210

Magenta, see Colour
mapMaybe, 223
margin, 68
Mark, see MarkState
markCheckControlItems, 196
marked, 216
markMenuItems, 230
MarkState, 216
  Mark, 216
NoMark, 216
maxFixedWindowSize, 258
MaxRGB, 240
maxScrollWindowSize, 258
MaxSize, 192
Maybe, 223
  Just, 223
Nothing, 223
MDI, see DocumentInterface
MDIProcess, 142
Menu, see menu
  menu, 13, 101, 103, 143, 151
    API, 20
      —attribute, 102, 229
        MenuFunction, 102, 229
        MenuId, 102, 229
        MenuIndex, 102, 229
MenuInit, 102, 229  
MenuMarkState, 102, 229  
MenuModsFunction, 102, 229  
MenuSelectState, 102, 229  
MenuShortKey, 103, 229  
—item, 104, 229  
Menu, 229  
PopUpMenu, 229  
radio—, 105, 229  
receiver, 126  
—separator, 104, 229  
sub—, 13, 106, 229  
subsetting, 109, 111  
Windows—, 108, 142  
MenuAttribute, see menu  
MenuElements, 102, 232  
getMenuElementType, 232  
menuElementToHandles, 232  
menuElementToHandles, see MenuElements  
MenuElementType, 229  
MenuFunction, see menu  
MenuId, see menu  
MenuIndex, see menu  
MenuItem, see menu  
MenuMarkState, see menu  
MenuModsFunction, see menu  
MenuRadioItem, 229  
Menus, 101, 224  
getMenuType, 224  
onMenu, 101, 224  
MenuSelectState, see menu  
MenuSeparator, see menu  
MenuShortKey, see menu  
MenuType, 229  
message, 125  
—passing, 133  
asynchronous, 127  
synchronous, 127, 128  
MinRGB, 240  
mmperinch, 258  
Modifiers, 62, 216  
ModifiersFunction, 216  
MouseDown, see MouseState  
MouseDrag, see MouseState  
MouseFunction, 64, 216  
MouseMove, see MouseState  
MouseState, 64, 216  
MouseDown, 216  
MouseDrag, 216  
MouseLost, 216  
MouseMove, 216  
MouseUp, 216  
MouseStateFilter, 62, 216  
MouseUp, see MouseState  
moveControlViewFrame, 196  
movePenPos, 234  
movePoint, 214  
moveWindowViewFrame, 271  
MState, 230  
NDI, see DocumentInterface  
new command, 110  
newlineChars, 258  
NewLS, 88, 108, 118, 206, 214  
NilCS, 214  
NilLS, 87, 107, 118, 119, 206, 214  
NoError, see ErrorReport  
noLS, 18, 214  
nolS, 214  
NoMark, see MarkState  
NoModifiers, 216  
NonProportionalFontDef, 34, 240  
Nothing, see Maybe  
Notice, 68  
notice, 67, 112, 113  
NoticeButton, 68  
NoWrap, see WrapMode  
nsend, 192  
nsend_MT, 192  
OffsetFun, 216  
OkBool, 216  
open command, 111  
openBitmap, 44, 39, 191  
openCompoundControls, 196  
openControls, 196  
openDefaultFont, 33, 234  
openDialog, see Dialogs  
openDialogFont, 33, 234  
opentFont, 32, 234  
opent, 32, 234  
openId, see Ids  
opentIds, see Ids  
onMenu, see Menus  
onMenuElements, 224  
onModalDialog, see Dialogs  
onNotice, 68  
onPopUpControlItems, 196  
onProcesses, see Processes  
onR2Id, see Ids  
onR2Ids, see Ids  
onRadioMenuItems, 224  
onReceiver, see Receivers
INDEX

openRId, see Ids
openRIds, see Ids
openSendNotifier, 210
openSubMenuElements, 224
openTCPListener, 260
openTimer, see Timers
openWindow, see Windows
OptionOnly, 216
oval, 37, 240
PageDimensions, see printing
PageDimensions, 242
ParentIndex, 216
paste command, 110
Pen
—Attribute, 240
  PenBack, 240
  PenColour, 240
  PenFont, 240
  PenPos, 240
  PenSize, 240
PgDownKey, see SpecialKey
PgUpKey, see SpecialKey
Pi, 39, 240
Picture, 29, 49, 78, 79
—Attribute, 30
  PicturePenBack, 31
  PicturePenColour, 31
  PicturePenFont, 31
  PicturePenPos, 30
  PicturePenSize, 30
PicturePenColour, see Picture
PicturePenFont, see Picture
PicturePenPos, see Picture
PicturePenSize, see Picture
pixel, 30, 162
playSoundFile, 256
point, 162
Point2, 214
Polygon, 42, 240
PolygonAt, 234
process, 15, 139
active, 141
API, 22
—attribute, 140, 216
  ProcessActivate, 141, 216
  ProcessClose, 141, 216
  ProcessDeactivate, 141, 216
  ProcessNoWindowMenu, 142, 216
  ProcessOpenFiles, 216
  ProcessOpenFiles, 141
  ProcessToolbar, 141
  ProcessToolbar, 216
  ProcessWindowPos, 141, 216
  ProcessWindowSize, 141, 216
  —group, 148
  —initialisation, 140
cancel dialogue, 159, 165
—anything, 157
defaultPrintSetup, 157
—dialogue, 161
—environment, 159
getPageDimensions, 160
—header, 167–169
JobInfo, 157, 158
margins, 100
origin, 160, 165
PageDimensions, 160
—parameters, 157
print, 158, 161
print job dialogue, 157
print setup dialogue, 157
PrintInfo, 158, 159
printPagePerPage, 164
PrintSetup, 157, 158
printText1, 165, 245
printText2, 167, 245
printText3, 168, 245
printUpdateFunction, 164
range, 159
scaling, see scaling
—text, 157, 165, 167
—trailer, 168, 169
printPagePerPage, 242, see printing
PrintSetup, see printing
PrintSetup, 242
printSetupDialog, 242
printSetupTypical, 258
printText1, see printing
printText2, see printing
printText3, see printing
printUpdateFunction, 242, see printing
Process, 249
ProcessActivate, 141, 216
ProcessClose, 141, 216
ProcessDeactivate, 141, 216
ProcessNoWindowMenu, 142, 216
ProcessOpenFiles, 216
ProcessOpenFiles, 141
ProcessToolbar, 141
ProcessToolbar, 216
ProcessWindowPos, 141, 216
ProcessWindowSize, 141, 216
—group, 148
—initialisation, 140
save dialogue, 159, 165
—anything, 157
defaultPrintSetup, 157
—dialogue, 161
—environment, 159
getPageDimensions, 160
—header, 167–169
JobInfo, 157, 158
margins, 100
origin, 160, 165
PageDimensions, 160
—parameters, 157
print, 158, 161
print job dialogue, 157
print setup dialogue, 157
PrintInfo, 158, 159
printPagePerPage, 164
PrintSetup, 157, 158
printText1, 165, 245
printText2, 167, 245
printText3, 168, 245
printUpdateFunction, 164
range, 159
scaling, see scaling
—text, 157, 165, 167
—trailer, 168, 169
printPagePerPage, 242, see printing
PrintSetup, see printing
PrintSetup, 242
printSetupDialog, 242
printSetupTypical, 258
printText1, see printing
printText2, see printing
printText3, see printing
printUpdateFunction, 242, see printing
Process, 249
ProcessActivate, 141, 216
Index

—window, 141
ProcessActivate, see process
ProcessClose, see process
ProcessDeactivate, see process
Processes, 142, 247
openProcesses, 142, 247
startProcesses, 142, 247
ProcessInit, 17, 140, 249
ProcessNoWindowMenu, see process
ProcessOpenFilesFunction, 216
ProcessWindowPos, see process
ProcessWindowResize, see process
ProcessWindowResizeFunction, 216
ProcessWindowSize, see process
PSt, 15, 143, 250

quit command, 111

R2Id, see identification
R2IdToId, 212
RadioControl, see control
RadioControlItem, 207
RadioMenu, see menu
Receive, 192
receive, 192
receive_MT, 192
Received, 192
ReceiveMsg, 192
Receiver, 125, 255
receiver, 13, 125, 132, 143, 147
API, 21
—attribute, 126, 255
ReceiverConnectedReceivers, 255
ReceiverInit, 126, 255
ReceiverSelectState, 126, 255
bi-directional, 125, 128, 135
—identification, 25, 125
uni-directional, 125, 127
Receiver2, 126, 135, 255
Receiver2Function, 255
ReceiverFunction, 255
receivers, 126, 252
getReceiverType, 252
openReceiver, 126, 252
ReceiverSelectState, see receiver
ReceiverType, 255
receiveUpTo, 192
receiving, see TCP
Rectangle, 40, 214
rectangleSize, 63, 214
RectangleToUpdateState, 216
Red, see Colour
redo command, 110
Region, 47, 234
toRegion, 47
[]], 47
PolygonAt, 47
Rectangle, 47
::, 47
reopenReceiver, see Receivers
resizeBitmap, 44, 191
resolution, 32
printer—, 160, 163
screen—, 159, 162, 163
—emulation, 159, 162
restorePos, see CharStreams
RGB, see Colour
RGBColour, 240
RId, see identification
RIdToId, 212
Right, see ItemPos
RightBottom, see ItemPos
RightJustify, see WrapMode
RightKey, see SpecialKey
RightTo, see ItemPos
RightTop, see ItemPos
RightToPrev, see ItemPos
Rows, see RowsOrColumns
RowsOrColumns, 77, 81, 207
Columns, 77, 81, 207
Rows, 77, 81, 207
SansSerifFontDef, 34, 240
save as command, 111
save command, 111
savePos, see CharStreams
scaling
explicit—, 162, 163
implicit—, 162, 164
screen emulation, see printing
ScrollFunction, 216
SDI, see Document Interface
SDIProcess, 143
selectChannel, see TCP
selectChannel_MT, 260
selectInputFile, see FileSelectEnv
selectOutputFile, see FileSelectEnv
selectPopUpControlItem, 196
selectRadioControlItem, 196
selectRadioMenuIndexItem, 230
selectRadioMenuItem, 230
SelectReceive, 260
accRChannels, 260
getRState, 260
SelectResult, 263
SelectSend, 260
accSChannels, 260
appDisconnected, 260
SelectState, 216
Able, 216
Unable, 216
Send, 192
send, 192
send_MT, 192
send_NB, 192
Sendable, 192
SendDeadlock, see SendReport
SendEvent, 192
sending, see TCP
SendNotifier, 263
SendOk, see SendReport
SendReport, 127, 252
SendDeadlock, 127
SendOk, 127
OtherSendReport, 252
SendDeadlock, 252
SendOk, 252
SendUnableReceiver, 252
SendUnknownReceiver, 252
SendUnableReceiver, 127
SendUnknownReceiver, see SendReport
SendUnknownReceiver, see SendReport
SerifFontDef, 34, 240
server, see TCP
setActiveControl, 271
setActiveWindow, 62, 271
setClipboard, 153, 195
setControlLook, 196
setControlLooks, 96, 196
setControlPos, 196
setControlScrollFunction, 196
setControlText, 130, 196
setControlTexts, 196
setControlViewDomain, 196
setDefaultPenBack, 234
setDefaultPenColour, 234
setDefaultPenFont, 234
setDefaultPenSize, 234
setEditControlCursor, 130, 196
setMaxSize, 192
setMenuElementCursor, 130, 196
setMenuTitle, 224
setMenuText, 224
setPenAttributes, 234
setPenBack, 234
setPenColour, 234
setPenFont, 234
setPenPos, 234
setPenSize, 234
setProcessActivate, 250
setProcessDeactivate, 250
setSliderState, 196
setSliderStates, 196
setSliderThumb, 196
setSliderThumbs, 196
setTimerInterval, 266
setWindowCancel, 271
setWindowCursor, 271
setWindowKeyboardStateFilter, 271
setWindowLook, 57, 63, 271
setWindowMouseStateFilter, 271
setWindowOk, 271
setWindowOuterSize, 271
setWindowPos, 271
setWindowScrollFunction, 271
setWindowTitle, 271
setWindowViewDomain, 271
setWindowViewSize, 271
ShiftOnly, 216
showControl, 196
showControls, 196
showProcess, 247
Size, 214
SliderAction, 216
SliderControl, see control
SliderDecLarge, see SliderMove
SliderDecSmall, see SliderMove
SliderIncLarge, see SliderMove
SliderIncSmall, see SliderMove
SliderMove, 83, 216
SliderDecLarge, 83, 216
SliderDecSmall, 83, 216
SliderIncLarge, 83, 216
SliderIncSmall, 83, 216
SliderThumb, 83, 216
SliderState, 82, 216
SliderThumb, see SliderMove
SmallFontDef, 34, 240
SpecialKey, 222, see KeyboardState
            BackSpaceKey, 222
            BeginKey, 222
            ClearKey, 222
            DeleteKey, 222
            DownKey, 222
            EndKey, 222
            EnterKey, 222
            EscapeKey, 222
            F10Key, 222
            F11Key, 222
            F12Key, 222
            F13Key, 222
            F14Key, 222
            F15Key, 222
            F1Key, 222
            F2Key, 222
INDEX

FSKey, 222
dotted decimal form, 172
F4Key, 222
dotted decimal notation, 173
F5Key, 222
duplex channel, 173
F6Key, 222
eom, 175
F7Key, 222
events, 186
F8Key, 222
flushBuffer_MT, 178
F9Key, 222
IP address, 172, 173
HelpKey, 222
listening, 172, 174
LeftKey, 222
lookupIPAddress, 173
PgDownKey, 222
nsend_MT, 178
PgUpKey, 222
port number, 172-174
RightKey, 222
receive_MT, 175
UpKey, 222
receiving, 174-176, 186

SR_Available, 263
selectChannelMT, 185
SR_Disconnected, 263
selectChannel_MT, 180
SR_Sendable, 263
send notifier, 188
stackWindow, 271
send_MT, 178
StandardCursor, see CursorShape
sending, 177, 178, 186
Start, 16
server, 172, 174, 180
StartPrinting, see Alternative
StdChannels, 173, 175
startIO, 16, 130, 142, 247
StdTCPChannels, 173
startProcesses, see Processes
StdTCPDef, 173
state
StringRChannel, 185
I/O—, 15
StringSChannel, 185
local—, 12, 111, 122
StringRCharStream, 184
process—, 12
StringSCharStream, 184
—transition, 12, 164
timeout, 173, 175, 178, 181
stdScrollFunction, 216
TCP_LISTENER, 263
stdUnfillNewFrameLook, 216
TCP_Listener, 174
stdUnfillUpdAreaLook, 216
TCP_RCharStream, 184
StringChannelReceiver, 257
TCP_SCharStream, 184
StringRChannel, 257
tcpPossible, 263
StringRCharStream, 257
TCP_RCHANNEL, 263
StringRChannels, 257
TCP_RCharStreams, 263
StringSChannel, 257
TCP_RCharStream, 263
StringSCharStream, 257
TCP_RCharStreams, 260
StringSChannels, 257
tcpSend, 127, 147, 151, 252
SubMenu, see menu
tcpSend2, 128, 135, 252
SymbolFontDef, 34, 240
TCP_171
text
draw, see drawing

TCP

drawing, see drawing
—metrics, 33
TextControl, see control
Tick, 265
tickDifference, 265
ticksPerSecond, 117, 119, 258
Time, 265
TimeEnv, 265
getBlinkInterval, 265
getCurrentDate, 265
getCurrentTick, 265
ggetCurrentTime, 265
Timeout, 192
timeout, see TCP
TimeoutReport, 192
Timer, see timer
timer, 13, 117, 147
    API, 21
        —attribute, 117, 268
            TimerFunction, 117, 268
            TimerId, 117, 268
            TimerInit, 117, 268
            TimerSelectState, 118, 268
receiver, 126
    Timer, 118, 268
TimerAttribute, see timer
TimerElements, 118, 269
    getTimerElementType, 269
timerElementTypeToHandles, 269
    timerElementToHandles, see TimerElements
TimerElementType, 268
TimerFunction, 268, see timer
TimerId, see timer
TimerInterval, 117, 268
Timers, 118, 266
    getTimerType, 266
    openTimer, 118, 266
TimerSelectState, see timer
TimerType, 268
Title, 214
toByteSeq, 263
toClipboard, see Clipboard
ToolBarItem, 216
    ToolbarItem, 216
        ToolbarSeparator, 216
    toCharStream, 260
toDomain, 234
toGBColour, 234
toHeaderStream, 260
toInputStream, 257
toOutputStream, 257
toVector, 214
TR_Expired, 192
TR_NoSuccess, 192
TR_Success, 192
trailer, see printing
    u_isJust, 223
    u_isNothing, 223
Unable, see SelectState
    undef, 133
UnderlinedStyle, 35, 240
undo command, 110
undraw, see Drawables
undrawAt, see Drawables
    unfill, see Fillables
    unfillAt, see Fillables
unmarkCheckControlItems, 196
    unmarkMenuItems, 230
UpdateArea, 216
    updateControl, 196
UpdateState, 56, 78, 79, 216
    updateWindow, 271
    UpKey, see SpecialKey
Vector2, 214
Vertical, see Direction
ViewDomain, 216
    viewDomainRange, 216
    ViewFrame, 216
    viewFrameRange, 216
vinch, 258
vmm, 258
Void, 214
wait, 265
White, see Colour
    WhiteRGB, 240
Window, 49, 95, 278
    window, 13, 49, 129
        active—, 51, 61, 112
    API, 20
        —attribute, 51, 278
            WindowActivate, 51, 61, 278
            WindowCancel, 54, 278
            WindowClose, 52, 278
            WindowCursor, 53, 278
            WindowDeactivate, 51, 61, 278
            WindowHeight, 52, 278
            WindowHeightScroll, 53, 278
            WindowId, 52, 278
            WindowIndex, 52, 278
            WindowInit, 52, 278
            WindowInitActive, 52, 278
            WindowItemSpace, 52, 278
            WindowKeyboard, 53, 62, 278
            WindowCursor, 53, 55, 164, 278
            WindowMouse, 53, 62, 278
            Windows, 54, 69, 278
            WindowOrigin, 53, 278
            WindowOuterSize, 52, 278
WindowPen, 53, 278
WindowPos, 52, 278
WindowSelectState, 53, 278
WindowViewDomain, 53, 55, 119, 278
WindowViewSize, 52, 278
WindowVMargin, 52, 278
WindowVScroll, 53, 278
control layer, 50, 60
document layer, 50, 55
—frame, 50, 60, 86
rendering
direct—, 55
indirect—, 55, 164
stacking order, 51
WindowActivate, see window
WindowAttribute, see window
WindowCancel, see window
WindowClose, see window
WindowCursor, see window
WindowDeactivate, see window
WindowHide, see window
WindowHMargin, see window
WindowHScroll, see window
WindowId, see window
WindowIndex, see window
WindowInit, see window
WindowItemSpace, see window
WindowKeyboard, see window
WindowLook, see window
WindowMinimumSize, see window
WindowMouse, see window
WindowOk, see window
WindowOrigin, see window
WindowPos, see window
WindowResize, see window
Windows, 54, 271
   getWindowType, 271
   openWindow, 54, 271
WindowSelectState, see window
WindowSize, see window
WindowType, 278
WindowViewDomain, see window
WindowVMargin, see window
WindowVScroll, see window
WrapMode, 165, 166, 245
   LeftJustify, 165, 166, 245
   NoWrap, 165, 166, 245
   RightJustify, 165, 166, 245
WState, 196

Yellow, see Colour