Pict Libraries Manual

Pict 4.1

Benjamin C. Pierce

Computer Science Department
Indiana University
Lindley Hall 215
Bloomington
Indiana 47405-4101
USA
pierce@cs.indiana.edu

David N. Turner

Department of Computing Science
University of Glasgow
Glasgow G12 8QQ
United Kingdom
dnt@dcs.gla.ac.uk

March 22, 1998

Contents

Copying

Pict is copyright ©1993–1997 by Benjamin C. Pierce and David N. Turner. This program and its documentation are free software; you can redistribute them and/or modify them under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. Pict is distributed in the hope that it will be useful, but *without any warranty*; without even the implied warranty of *merchantability* or *fitness for a particular purpose.* See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Chapter 1

Library Conventions

The Pict language does not impose any restrictions on the form of a library module (a separately-compiled piece of Pict code can be any well-typed sequence of top-level declarations). However, by convention, all our library modules have the following structure:

```
val [
  #X1
  ...
  #Xn
  module : [
    op1 = T1
    ...
    opn = Tn
  ]
] = ...
```

Thus, each library module introduces a (possibly empty) set of top-level type bindings, $X_1, \ldots, X_n$, which may either be abstract or opaque types, and a single top-level value identifier `module`, which is bound to some record of operations $op_1, \ldots, op_n$. The value identifier is usually the lowercase version of the library filename (for example, `Std/Array` introduces the value identifier `array`).

When we document a module, we only display the type bindings and the types of all operations (interspersed with text explaining the operations):

```
  #X1
  ...
  #Xn
  op1 = T1
  ...
  opn = Tn
```

For example, after importing `Std/Array`, the type `Array` is directly available to the programmer and the value identifier `array` is bound to a record of operations on arrays. This allows us to use qualified names for our array operations. For instance, `array.size` is a function which returns the size of an array.

1.1 Naming Conventions

When naming operations, we have tried to be as uniform as possible. We use the following naming conventions:

`==`, `<>`, `<<`, `>>`, `<=`, `>=` Comparison functions.

`size` The returns the size or length of an object. For example `list.size` returns the length of a list, and `string.size` returns the number of characters in a string.

`pr` Prints an object to stdout. For example, `(int.pr x)` prints the integer `x` to stdout.

`hash` Computes a hash value for an object. For example, `(string.hash s)` computes a has value for the string `s`.

`rev` Reverses the order of objects in an aggregate object. For example, `(list.rev l)` reverses the list `l`.

`append` Appends two aggregate objects.

`apply` Applies a function to each element of an aggregate object. For example, `(list.apply l f)` applies `f` to each element of the list `l`.

`revApply` Same as `apply`, but traverses the aggregate object in the opposite order (for aggregate values which do not have a useful ordering on their elements, such as hashtables, only the `apply` operation is provided).

`itApply` Similar to `apply`, except that the index of each element is also supplied to the user function.

`revItApply` Similar to `itApply`, except that it traverses the aggregate object in the opposite order.

`fold` Accumulates a result by applying a user-specified function to each element of an aggregate object.

`revFold` Same as `fold`, but traverses the aggregate object in the opposite order (for aggregate values which do not have a useful ordering on their elements, such as hashtables, only the `fold` operation is provided).

`itFold` Similar to `fold`, except that the index of each element is also supplied to the user function.

`revItFold` Similar to `itFold`, except that it traverses the aggregate object in the opposite order.

`map` Applies a user-specified function to each element of an aggregate object, returning a new aggregate object.

`revMap` Similar to `map`, except that it traverses the aggregate object in the opposite order.

`cmp` Compares two objects. For example, `(string.cmp s t)` returns a single value which indicates the relative string ordering of `s` and `t`.

`make` Builds an instance of an object. For example, `array.make` builds a fresh array (given the size of the array and an initialising element).

`nth` Extracts the `n`'th element of an aggregate object. For example, `(string.nth s i)` extracts the `i`'th character of `s`.

`tabulate` Builds an instance of an object by calling a given function to generate elements. For example, `(list.tabulate n f)` builds a list of size `n` where the `i`'th element of the list is initialised to `(f i)`

`toString` Converts an object to a string. For example, `int.toString` converts an integer to a string.

`fromString` Converts an string to an object. For example, `int.fromString` converts a string to an integer.

`unsafe...` Unsafe version of an operation which usually does bounds checking. For example, `array.unsafeUpdate` updates an array, just like `array.update`, but it avoids doing any bounds checking. Unsafe operations should be used with care, since they can cause catastrophic errors in the runtime system if given out-of-range arguments.

1.2 Library Groups

There are five groups of library modules:

`Std` Standard library modules which we expect them to exist on any platform which supports Pict. Other sets of modules, such as the `Posix` modules, may not exist on all systems.

`Posix` Interfaces to Posix facilities. We have tended to add support for Posix facilities as and when we have required them, so we support all Posix functionality.

`X11` Low-level interfaces to `X11` facilities.

`Picture` A high-level library of composable pictures which is built in terms of `X11` primitives.

`Contrib` Various bits and pieces.

Chapter 2

The Pervasive Environment

To avoid having to write explicit `import` statements for commonly-used library modules, the Pict compiler always imports the following modules before compilating any Pict program (this behaviour can be disabled by resetting the `lib` flag).

```
import "Std/Bool"
import "Std/Cmp"
import "Std/Exn"
import "Std/Int"
import "Std/List"
import "Std/Misc"
import "Std/Opt"
import "Std/String"
```

Chapter 3

Standard Libraries

3.1 `Std/Array`: Array Utilities

3.1.1 Types

```
#Array : (Type -> Type)
```

`(Array X)` is the type of arrays which contain elements of type `X`.

3.1.2 Creation

```
empty = /[#X /(Array X)]
```

Returns an array of length zero. This function is useful since, unlike `make`, it does not require an initializing element.

```
make = /[#X X Int /(Array X)]
```

`(make x n)` returns a new array containing `n` elements, each initialized to `x`. The array size `n` must be greater than or equal to zero.

```
tabulate = /[#X Int /[Int /X] /(Array X)]
```

`(tabulate n f)` creates an array of size `n` where each array element `i` is initialised to `(f i)`. The function `f` is called sequentially and in order. `n` must be greater than or equal to zero.

```
fromList = /[#X (List X) /(Array X)]
```

`(fromList l)` creates an array from the list `l`.

3.1.3 Interrogation

```
size = /[#X (Array X) /Int]
```

`(size a)` returns the size of the array `a`.

```
nth = /[#X (Array X) Int /X]
unsafeNth = /[#X (Array X) Int /X]
```

`(nth a n)` Looks up the `n`'th element of the array `a`, if $0 \leq$ `n` $<$ `(size a)`. If `n` is not a valid index for `a` then we generate a runtime error. `unsafeNth` behaves similarly, except that it does not do any range checking.

3.1.4 Modification

```
update = /[#X (Array X) Int X Sig]
unsafeUpdate = /[#X (Array X) Int X Sig]
```

(`update a n x`) updates the `n`'th element of `a` to be `x`. If `n` is not a valid index for `a` then we generate a runtime error. `unsafeUpdate` behaves similarly, except that it does not do any range checking.

```
reset = /[#X (Array X) X Sig]
```

(`reset a x`) sets the value of each element in `a` to `x`.

```
rotate = /[#X (Array X) Int Sig]
```

(`rotate a o`) rotates `a` so that the `i`'th element of `a` contains the value which used to be at index `i+o`. For example, the expression (`rotate a -1`) rotates the contents of `a` to the right by one.

3.1.5 Iteration

```
fold = /[#X #Y (Array X) /[Int X Y /Y] Y /Y]
revFold = /[#X #Y (Array X) /[Int X Y /Y] Y /Y]
```

(`fold a f init`) applies `f` to each element of `a` (plus its index), accumulating a result of type `Y`. `f` is called sequentially and in order, with the initial accumulated argument being `init`. `revFold` behaves similarly, except that it traverses the array in reverse order.

```
apply = /[#X (Array X) /[Int X Sig] Sig]
revApply = /[#X (Array X) /[Int X Sig] Sig]
```

(`apply a f`) applies `f` to each element of `a`, along with its index. `f` is called sequentially and in order. `revApply` behaves similarly, except that it traverses the array in reverse order.

```
map = /[#X (Array X) /[Int X /X] Sig]
revMap = /[#X (Array X) /[Int X /X] Sig]
```

(`map a f`) applies `f` to each element of `a`, along with its index, updating each element in `a` with the result of applying `f`. `f` is called sequentially and in order. `revMap` behaves similarly, except that it traverses the array in reverse order.

3.2 `Std/Array2`: 2D Array Utilities

3.2.1 Types

```
#Array2 : (Type -> Type)
```

(`Array2 X`) is the type of two dimensional arrays which contain elements of type `X`.

3.2.2 Creation

```
empty = /[#X /(Array2 X)]
```

(`empty`) returns a 2D array of size zero. This function is useful since, unlike `make`, it does not require an initializing element.

```
make = /[#X X Int Int /(Array2 X)]
```

(`make x w h`) returns a new array of dimension `[w h]`, with each element initialized to `x`. Both the width `w` and height `h` must be greater than or equal to zero.

```
tabulate = /[#X Int Int /[Int Int /X] /(Array2 X)]
```

(`tabulate w h f`) creates an array of size `[w h]` where each element `[x y]` is initialised to (`f x y`). Both `w` and `h` must be greater than or equal to zero. The function `f` is called sequentially, and in order (`[0 0]`, `[0 1]`, ..., `[0 (h-1)]`, `[1 0]`, `[1 1]`, ..., `[w h]`).

```
fromList = /[#X (List(List X)) /(Array2 X)]
```

(`fromList l`) creates an array from a list of lists of elements. The sub-lists of `l` must all be of the same length (since `Array2` only allows rectangular arrays). If this is not the case, we generate a runtime error.

3.2.3 Interrogation

```
size = /[#X (Array2 X) /[Int Int]]
```

(`size a`) returns the width and height of `a`.

```
nth = /[#X (Array2 X) Int Int /X]
unsafeNth = /[#X (Array2 X) Int Int /X]
```

(`nths a x y`) looks up the element at position `[x y]` in `a`. It must be the case that `[0 0]` $\leq$ `[x y]` and `[x y]` $<$ (`size a`), otherwise we generate a runtime error. `unsafeNth` behaves similarly, except that it does not do any range checking.

3.2.4 Modification

```
update = /[#X (Array2 X) Int Int X Sig]
unsafeUpdate = /[#X (Array2 X) Int Int X Sig]
```

(`update a x y v`) updates the element at `[x y]` in `a` to be `v`. If `[x y]` is not a valid index for `a` then we generate a runtime error. `unsafeUpdate` behaves similarly, except that it does not do any range checking.

```
reset = /[#X (Array2 X) X Sig]
```

(`reset a x`) sets the value of each array element to `x`.

```
rotate = /[#X (Array2 X) Int Int Sig]
```

The expression (`rotate a dx dy`) rotates `a` so that the `[i j]`'th element of `a` contains the value which used to be stored at index `[(+ i dx) (+ j dy)]`. For example, the expression (`rotate a -1 -1`) shifts the contents of `a` up and to the right by one.

3.2.5 Iteration

```
fold = /[#X #Y (Array2 X) /[Int Int X Y /Y] Y /Y]
revFold = /[#X #Y (Array2 X) /[Int Int X Y /Y] Y /Y]
```

(`fold a f i`) applies `f` to each element of `a` (plus its index), accumulating a result of type `Y`. `f` is called sequentially, and in order, with the initial accumulated argument being `i`. `revFold` behaves similarly, except that it iterates over `a` in reverse order.

```
apply = /[#X (Array2 X) /[Int Int X Sig] Sig]
revApply = /[#X (Array2 X) /[Int Int X Sig] Sig]
```

(`apply f a`) applies `f` to each element of `a`, along with its index. `f` is called sequentially, and in order. `revMap` behaves similarly, except that it iterates over `a` in reverse order.

```
map = /[#X (Array2 X) /[Int Int X /X] Sig]
revMap = /[#X (Array2 X) /[Int Int X /X] Sig]
```

(`map a f`) applies `f` to each element of `a`, along with its index, updating each element in `a` with the result of applying `f`. `f` is called sequentially, and in order. `revMap` behaves similarly, except that it iterates over `a` in reverse order.

3.3 Std/Bool: Booleans

3.3.1 Types

The type Bool is built in, and cannot be redefined. The boolean constructors true and false are also built in, and cannot be redefined.

3.3.2 Operations

```
&&  = /[Bool Bool /Bool]
||  = /[Bool Bool /Bool]
xor = /[Bool Bool /Bool]
not = /[Bool /Bool]
```

Conjunction, disjunction, exclusive-or and negation of booleans.

3.3.3 Iteration

```
while = /[/[/Bool] /[Sig] Sig]
until = /[/[Sig] /[/Bool] Sig]
```

(while b f) evaluates (f) while (b) is true. Similarly, (until f b) evaluates (f) until (b) is true.

3.3.4 Conversion

```
fromString = /[String Exn /Bool]
```

(fromString s h) converts the string s to an boolean. If s is not equal to the string "true" or the string "false" then error handler h is called.

```
toString = /[Bool /String]
```

(toString b) converts b to a string.

```
fromInt = /[Int /Bool]
```

(fromInt x) converts the integer x to an boolean. If x is zero, we return false, otherwise we return true.

```
toInt = /[Bool /Int]
```

(toInt b) converts b to the integer 1 if b is true, and 0 if b is false.

3.3.5 Printing

```
pr = /[Bool Sig]
```

(pr b) prints b on stdout.

3.3.6 Abbreviations

```
val &&    = bool.&&
val ||    = bool.||
val xor   = bool.xor
val not   = bool.not
val while = bool.while
val until = bool.until
```

3.4 Std/Bytes: Byte Arrays

3.4.1 Types

```
#Bytes
```

We provide the type `Bytes` since an array of bytes can be represented much more efficiently than array of type `(Array Char)`.

3.4.2 Creation

```
empty = /[/Bytes]
```

Returns an array of bytes of length zero.

```
make = /[Char Int /Bytes]
```

`(make c n)` returns a new array containing `n` bytes, each initialized to `c`. The array size `n` must be greater than or equal to zero.

```
extend = /[Bytes Char Int /Bytes]
```

`(extend b c n)` creates a new byte array which contains all the elements of `b`, followed by `n` copies of the character `c`.

```
fromString = /[String /Bytes]
```

Creates a byte array from the given string.

3.4.3 Interrogation

```
size = /[Bytes /Int]
```

`(size b)` returns the size of the bytearray `b`.

```
nth = /[Bytes Int /Char]
unsafeNth = /[Bytes Int /Char]
```

`(nth b n)` looks up the `n`'th element of the bytearray `b`, if `0` $\leq$ `n` $<$ `(size b)`. If `n` is not a valid index for `b` then we generate a runtime error. `unsafeNth` behaves similarly, except that it does not do any bounds checking.

3.4.4 Modification

```
update = /[Bytes Int Char Sig]
unsafeUpdate = /[Bytes Int Char Sig]
```

`(update b n c)` updates the `n`'th element of `b` to be `c`. If `n` is not a valid index for `b` then we generate a runtime error. `unsafeUpdate` behaves similarly, except that it does not do any bounds checking.

```
copy = /[src=Bytes srcIdx=Int len=Int dst=Bytes dstIdx=Int Sig]
```

`(copy src=s srcIdx=i len=l dst=s dstIdx=j)` copies `l` bytes from `s` (starting from `i`) into `d` (starting from `j`). `i` and `j` must be valid indexes for `s` and `d`, `l` must be greater than or equal to zero, `(+ i l)` and `(+ j l)` must be less than or equal to `(size s)` and `(size d)` respectively, otherwise we generate a runtime error.

```
copyFromString =
  /[src=String srcIdx=Int len=Int dst=Bytes dstIdx=Int Sig]
```

`(copyFromString src=s srcIdx=i len=l dst=d dstIdx=j)` copies `l` bytes from `s` (starting from `i`) into `d` (starting from `j`). `i` and `j` must be valid indexes for `s` and `d`, `l` must be greater than or equal to zero, `(+ i l)` and `(+ j l)` must be less than or equal to `(string.size s)` and `(size d)` respectively, otherwise we generate a runtime error.

3.4.5 Conversion

```
toString = /[Bytes idx=Int len=Int /String]
```

(`toString b idx=i len=l`) extracts a string of length `l` from `b`, starting at `i`. We generate a runtime error if `i` is not a valid index for `b`, or if `(+ i l)` is greater than `(size b)`.

```
unsafeCast = /[Bytes /String]
```

Exposes the internal representation of a bytearray. This function is only intended to be used in other library modules, and should not be relied upon in user programs (since the representation of bytearrays may well change in the future).

3.5 `Std/Char`: Characters

3.5.1 Types

The type `Char` is builtin and is a subtype of `Int`. This means that integer arithmetic operations and comparisons will also accept characters as arguments. The integer value of each character is its ASCII code.

3.5.2 Character Classification

These functions check whether a character falls into a certain character class (according to the current locale).

```
isAlnum = /[Char /Bool]
```

Checks for an alphanumeric character; it is equivalent to `(|| (isAlpha c) (isDigit c))`.

```
isAlpha = /[Char /Bool]
```

Checks for an alphabetic character; it is equivalent to `(|| (isUpper c) (isLower c))`.

```
isCntrl = /[Char /Bool]
```

Checks for a control character.

```
isDigit = /[Char /Bool]
```

Checks for a digit.

```
isGraph = /[Char /Bool]
```

Checks for any printable character except space.

```
isLower = /[Char /Bool]
```

Checks for a lower-case character.

```
isPrint = /[Char /Bool]
```

Checks for any printable character including space.

```
isPunct = /[Char /Bool]
```

Checks for any printable character which is not a space or an alphanumeric character.

```
isSpace = /[Char /Bool]
```

Checks for white-space characters. In the `"C"` and `"POSIX"` locales, these are: space, form-feed, newline, carriage return, horizontal tab, and vertical tab.

```
isUpper = /[Char /Bool]
```

Checks for an uppercase letter.

```
isXDigit = /[Char /Bool]
```

Checks for a hexadecimal digits, i.e. one of `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `0`, `a`, `b`, `c`, `d`, `e`, `f`, `A`, `B`, `C`, `D`, `E`, `F`.

3.5.3 Conversion

```
toUpper = /[Char /Char]
toLower = /[Char /Char]
```

`toUpper` converts `c` to upper case, if possible. `toLower` converts `c` to lower case, if possible. The character `c` is returned unchanged if the conversion was not possible.

```
fromInt = /[Int /Char]
unsafeFromInt = /[Int /Char]
```

Convert an ASCII code to a character. If `x` is not a valid ASCII code, then we generate a runtime error. `unsafeFromInt` behaves similarly, but it does not do any range checking.

```
toString = /[Char /String]
```

`(toString c)` creates a string of size one, containing the character `c`.

```
hash = /[Char /Int]
```

`(hash c)` returns a hash value for `c`.

3.5.4 Printing

```
pr = /[Char Sig]
```

`(pr c)` prints the character `c` on stdout.

3.6 `Std/Cmp`: Comparison Results

3.6.1 Types

```
#Cmp
```

Values of type `Cmp` are usually returned by comparison functions. They are particularly useful in the case of comparisons of strings or lists, since a single comparison operation, which returns a value of type `Cmp`, can be much more efficient that making two calls to an ordering predicate.

3.6.2 Construction

```
LT = Cmp
EQ = Cmp
GT = Cmp
```

The values `LT`, `EQ` and `GT` indicate that a comparison operator found that its first argument was (respectively) strictly less than, equal to, or strictly greater than its second argument.

3.6.3 Operations

```
lt = /[Cmp /Bool]
le = /[Cmp /Bool]
eq = /[Cmp /Bool]
ne = /[Cmp /Bool]
ge = /[Cmp /Bool]
gt = /[Cmp /Bool]
```

The above functions test whether the result of a comparison was strictly less than, less than or equal to, equal to, not equal to, greater than or equal to, or strictly greater than.

3.6.4 Conversion

```
toString = /[Cmp /String]
```

(`toString c`) converts `c` to a string.

3.6.5 Printing

```
pr = /[Cmp Sig]
```

(`pr c`) prints `c` on stdout.

3.7 `Std/Error`: Error Handling

This module provides some simple facilties to check for and signal fatal errors (i.e. errors which cause the whole Pict runtime system to exit).

```
exit = /[Int /Bot]
```

(`exit n`) terminates the whole runtime system, returning the error code `n`. This should only be used for fatal errors since it terminates all Pict processes, not just the current thread. `exit` never returns, so its result can be given any type.

```
fatal = /[String /Bot]
```

(`fatal s`) prints `s` on the standard error and then calls `exit` with error code `1`.

```
fail = /String
```

`fail` is similar to `fatal`, except that it does not require a result channel.

```
check = /[String Bool /[]]
```

(`check s b`) checks that the boolean expression `b` is true. If `b` is not true, then we print the string `s` and generate a runtime error. This function is used to implement various runtime checks such as array bounds checking. It is possible to disable these runtime checks by resetting the `checks` compiler flag.

```
assert = /[String Bool /[]]
```

`assert` is similar to `check`, except that by default it does *not* check that the condition `b` is true. It is possible to enable checking of assertions by setting the `assertions` compiler flag. This function is used in various 'unsafe' library calls which do not want to do range checking. Setting the `assertions` compiler flag enables us to conveniently enable all possible runtime checks.

3.8 `Std/Events`: Events

3.8.1 Types

```
#Event
#EChan : (Type -> Type)
```

The type (`EChan T`) describes "input event channels" from which elements of `T` can be read using `read`. The type `Event` is the type of "potential communcations."

3.8.2 Operations

```
make = /[#X /[!X (EChan X)]]
```

(`make`) creates a fresh output channel and an input event channel, and returns the two as a pair.

```
read = /[#X (EChan X) /X /Event]
```

(`read r receiver`) builds an event that, when triggered (with `sync`), will attempt to read a value from the event channel `r` and pass it on to `receiver`.

```
choose = /[Event Event /Event]
```

(`choose e1 e2`) builds an event that, when triggered, will attempt to perform both `e1` and `e2` at the same time. When either `e1` or `e2` succeeds, the other is aborted and the choose-event itself succeeds.

```
null  = Event
```

An "inert event" that never succeeds.

```
sync = /Event
```

(`sync e`) triggers the event `e`.

3.9 `Std/Exn`: Exception Handling

The Pict language provides no built-in facilities for raising or handling runtime exceptions. This library provides some simple utilities for programming with "exception continuations."

3.9.1 Types

```
#Handler = \X:Neg Type = /X
#Exn = (Handler String)
```

The type `(Handler X)` is just a synonym for the channel type `/X`. Values of type `(Handler X)` are used, by convention, to signal errors. The type `Exn` is just a convenient abbreviation for a handler which carries a string value.

3.9.2 Operations

```
fail = /[String /(Handler Top)]
```

(`fail s`) returns an exception handler which, if invoked, printing the error message `s` and exits the program.

```
ignore = (Handler Top)
```

A predefined handler that ignores any exceptions.

```
exit = Exn
```

A predefined handler that handles a string-carrying exception by printing the string and exiting the program.

```
tag = /[Exn String /Exn]
```

Given a string-carrying handler `h`, (`tag h s`) returns a handler that catches exceptions, appends the comment string `s`, and reraises them along `h`.

```
rename = /[Exn String /Exn]
```

Given a string-carrying handler `h`, `(rename h s)` returns a handler that catches exceptions, and reraises them along `h` using the given string `s`.

```
raise = /[#X (Handler X) X /Bot]
```

`(raise h x)` sends the value `x` to the handler `h` and blocks the caller. (The result type is `Bot` since `raise` never returns control to the caller.)

```
suspend = /[#X #Y (Handler [X /Y]) X /Y]
```

`(suspend h x)` sends `x` to the handler `h` and blocks the caller. The caller's continuation is included in the value sent to `h` so that the handler can continue from the point where the exception is raised, if it chooses.

```
try = /[#X #Y /[(Handler X) /Y] /[X /Y] /Y]
```

The expression `(try \(h) = body \(x) = default)` evaluates `body`, binding `h` to a new exception handler. If the execution of `body` is successful, we simply return the value (of type `Y`) returned by `body`. If, however, `h` is raised in `body`, then we pass the raised value (of type `X`) to the `default` expression.

3.10 `Std/External`: External Values

3.10.1 Types

```
#External : (Type -> Type)
```

Values of type `(External X)` may be safely stored outside the Pict heap.

3.10.2 Creation

```
make = /[#X X /(External X)]
```

`(make x)` informs the runtime system that the value `x` is referenced from outside the Pict heap. Unlike the value `x`, which may be moved during garbage collection, the value returned by `make` resides at a constant address, and can therefore be safely stored in C data structures.

3.10.3 Operations

```
get = /[#X (External X) /X]
```

`(get e)` extracts a Pict value from an external value `e`.

```
delete = /[#X (External X) Sig]
```

Any external values which are no longer required should be removed using `delete` (otherwise such values will *never* be reclaimed by the garbage collector). This function must be used with care, since anything can happen if one deletes an external value more than once, or attempts to show a deleted external value.

3.11 `Std/Float`: Floating Point Numbers

This module provides a simple implementation of single-precision floating point numbers. All floating point numbers are currently heap-allocated, so don't expect the performance of floating pointer operations to be that great (we hope to fix this sometime in the future, but don't hold you breath).

3.11.1 Types

```
#Float
```

3.11.2 Arithmetic Operations

```
+ = /[Float Float /Float]
- = /[Float Float /Float]
```

Addition and subtraction.

```
* = /[Float Float /Float]
div = /[Float Float /Float]
```

Product and quotient.

```
neg = /[Float /Float]
dec = /[Float /Float]
inc = /[Float /Float]
```

Negation, predecessor and successor.

3.11.3 Trigonometric Operations

```
sin = /[Float /Float]
cos = /[Float /Float]
tan = /[Float /Float]
asin = /[Float /Float]
acos = /[Float /Float]
atan = /[Float /Float]
```

Sine, cosine, tangent, arc sine, arc cosine, arc tangent.

3.11.4 Comparison

```
max = /[Float Float /Float]
min = /[Float Float /Float]
```

Maximum and minimum.

```
== = /[Float Float /Bool]
<> = /[Float Float /Bool]
```

Equality, inequality.

```
>> = /[Float Float /Bool]
<< = /[Float Float /Bool]
```

Strictly greater than, and strictly less than.

```
>= = /[Float Float /Bool]
<= = /[Float Float /Bool]
```

Greater than or equal to, and less than or equal to.

```
cmp = /[Float Float /Cmp]
```

`(cmps x y)` returns a single value indicating the ordering of `x` and `y` (cf. Section 3.6).

3.11.5 Conversion

```
fromInt = /[Int /Float]
```

(`fromInt x`) converts the integer `x` to a float.

```
fromString = /[String Exn /Float]
```

(`fromString s h`) converts a string `s` to a float. If `s` is not a valid string (a non-empty sequence of digits, possible prefixed by a negation symbol `~`) then `fromString` calls the error handler `h`.

```
toString = /[Float /String]
```

(`toString x`) converts `x` to a string.

```
hash = /[Float /Int]
```

(`hash x`) returns a hash value for `x`.

3.11.6 Printing

```
pr = /[Float Sig]
```

(`pr x`) prints `x` on stdout.

```
format = /[String Float Sig]
```

(`format s x`) calls the C `printf` function to format the integer `x` as specified in the string `s`. No checking of `s` is attempted by the Pict compiler, so be careful with this function.

3.12 `Std/Hash`: Hashtables

3.12.1 Types

```
#Hash : (Type Type -> Type)
```

(`Hash K V`) denotes the type of hashtables with keys of type `K` and elements of type `V`. We allow only fixed-size hashtables for the moment. This implementation will be extended later to allow dynamic resizing of hashtables.

3.12.2 Creation

```
make = /[#K #V /[K /Int] /[K K /Bool] /(Hash K V)]
```

(`make hash eq`) creates a new hashtable with the hash function `hash` and equality function `eq`.

3.12.3 Interrogation

```
lookup = /[#K #V (Hash K V) K /V]
```

(`lookup h k`) looks up the value associated with `k` in `h`. If `k` has no associated value, then we generate a runtime error.

```
tryLookup = /[#K #V (Hash K V) K /(Opt V)]
```

(`tryLookup h k`) returns (`opt.yes v`) if `v` is the value associated with `k` in `h`. If `k` has no associated value, it returns `opt.no`.

```
lookupDefault = /[#K #V (Hash K V) K V /V]
```

(`lookupDefault h k dflt`) looks up the value associated with `k` in `h`. If `k` has no associated value, then we return the default value `dflt`.

3.12.4 Modification

```
insert = /[#K #V (Hash K V) K V Sig]
```

`(insert h k v)` inserts the value `v` with key `k` into the hashtable `h`. This does not overwrite previous entries with the same key, so that a subsequent `(remove h k)` call will restore the previous value associated with `k`.

```
remove = /[#K #V (Hash K V) K Sig]
```

`(remove h k)` removes the value with key `k` from the hashtable. If no such value exists, we generate a runtime error.

3.13 `Std/Int`: Integers

3.13.1 Types

The type `Int` is built in, and cannot be redefined.

3.13.2 Arithmetic Operations

```
+ = /[Int Int /Int]
- = /[Int Int /Int]
```

Addition and subtraction.

```
* = /[Int Int /Int]
div = /[Int Int /Int]
mod = /[Int Int /Int]
```

Product, quotient and modulus. `div` and `mod` generate a runtime error if the divisor is zero. Note that, unlike the standard mathematical operations, `div` and `mod` round towards zero, so be careful if you intend to pass `div` and `mod` negative arguments.

```
neg = /[Int /Int]
dec = /[Int /Int]
inc = /[Int /Int]
```

Negation, predecessor and successor.

```
sgn = /[Int /Int]
```

`(sgn x)` returns `1` if `x` is strictly greater than zero, `0` if `x` equals zero, and `-1` if `x` is strictly less than zero.

```
abs = /[Int /Int]
```

`(abs x)` returns the absolute value of `x`.

3.13.3 Bitwise Operations

```
land = /[Int Int /Int]
lor = /[Int Int /Int]
lxor = /[Int Int /Int]
lnot = /[Int /Int]
```

Bitwise and, or, exclusive-or and negation.

```
bitIsSet = /[bit=Int Int /Bool]
```

`(bitIsSet bit=i x)` returns `true` if the `i`'th bit of the boolean representation of `x` is set.

3.13.4 Comparison

```
max = /[Int Int /Int]
min = /[Int Int /Int]
```

Maximum and minimum.

```
== = /[Int Int /Bool]
<> = /[Int Int /Bool]
```

Equality, inequality.

```
>> = /[Int Int /Bool]
<< = /[Int Int /Bool]
```

Strictly greater than, and strictly less than.

```
>= = /[Int Int /Bool]
<= = /[Int Int /Bool]
```

Greater than or equal to, and less than or equal to.

```
cmp = /[Int Int /Cmp]
```

`(cmp x y)` returns a single value indicating the ordering of `x` and `y` (cf. Section 3.6).

3.13.5 Iteration

```
apply = /[Int Int Int /[Int Sig] Sig]
```

`(apply start inc finish f)` applies `f` to the sequence of integers `start`, `start+inc`, ... until it reaches an integer strictly greater than `finish`, or strictly less than `finish` if `inc` is negative.

```
fold = /[#R Int Int Int /[Int R /R] R /R]
```

`(fold start inc finish f init)` applies `f` to the sequence of integers `start`, `start+inc`, ... until it reaches an integer strictly greater than `finish`, or strictly less than `finish` if `inc` is negative. Successive applications of `f` accumulate a result of type `R`, with `init` being the initial accumulated result.

```
for = /[Int Int /[Int Sig] Sig]
```

The `for` function is specialization of the `apply` function. The expression `(for min max f)` applies `f` to the integers `min`, ..., `max`.

3.13.6 Conversion

```
fromString = /[String Exn /Int]
```

`(fromString s h)` converts a string `s` to an integer. If `s` is not a valid string (a non-empty sequence of digits, possible prefixed by a negation symbol `~`) then `fromString` calls the error handler `h`.

```
toString = /[Int /String]
```

`(toString x)` converts `x` to a string.

```
hash = /[Int /Int]
```

`(hash x)` returns a hash value for `x`.

3.13.7 Printing

```
pr = /[Int Sig]
```

(pr x) prints x on stdout.

```
format = /[String Int Sig]
```

(format s x) calls the C printf function to format the integer x as specified in the string s. No checking of s is attempted by the Pict compiler, so be careful with this function.

3.13.8 Abbreviations

```
val + = int.+
val - = int.-
val inc = int.inc
val dec = int.dec
val neg = int.neg
val * = int.*
val div = int.div
val mod = int.mod
val == = int.==
val <> = int.<>
val >> = int.>>
val << = int.<<
val >= = int.>=
val <= = int.<=
val max = int.max
val min = int.min
val $$ = int.toString
```

3.14 Std/Io: Input/Output

This library implements a high-level non-blocking input/output abstraction.

3.14.1 Input Streams

```
#Rd =
  [eof = /[/Bool]
   read = /[Int Exn /String]
   readChar = /[Exn /Char]
   peekChar = /[Exn /Char]
   readBytes = /[Bytes Int Int /Int]
   close = /[Sig]]
```

The type Rd describes character stream objects for which we have read capability. The eof method says whether we have reached the end of file. The (read h n) method reads n characters from the file and returns them as a string. If there are not n characters available before the end of file, (read h n) signals an exception on the handler h. readChar reads the next character from the stream, or calls the given handler if we have reached the end of file. peekChar is similar, except that it does not remove the character from the buffer. (readBytes buff i n) reads at most n characters into buff, starting at offset i, and returns the number of characters read. (The number of characters actually read depends on the number available from the file at this moment: if there are fewer than n available now but the file is not at the eof marker, readBytes just reads those that are available now.) The close method closes the file. Calling any stream methods after a stream has been closed will cause a deadlock.

3.14.2 Output Streams

```
#Wr =
  [write = /[String Sig]
   writeChar = /[Char Sig]
   writeSubstring = /[String Int Int Sig]
   writeBytes = /[Bytes Int Int Sig]
   close = /[Sig]]
```

The type `Wr` describes character stream objects for which we have write capability. The expression `(write s)` writes the string `s` on the stream. The `writeChar` writes a character on the stream. `(writeSubstring s i n)` writes `n` characters beginning from position `i` in the string `s` (it must be the case that that `i` and `n` denote a valid substring of `s`). The expression `(writeBytes buff i n)` writes `n` characters from `buff` beginning from position `i` (it must be the case that that `i` and `n` denote a valid subarray of `buff`). The `close` method closes the stream. Calling any stream methods after a stream has been closed will cause a deadlock.

3.14.3 Operations

```
fdRd = /[Fd /Rd]
stringRd = /[String /Rd]
openRd = /[String Exn /Rd]
```

The function `fdRd` takes a file descriptor (which must be be opened for reading) and creates a buffered stream for that descriptor. `stringRd` takes a string a treats it as a stream. `openRd` takes a filename `n` and creates a stream connected to that file, calling the given handler if the file could not be opened.

```
fdWr = /[Fd /Wr]
```

The function `fdWr` takes a file descriptor (which must be be opened for writing) and creates a buffered stream for that descriptor.

```
openGenWr = /[String Bool Bool Bool Perm Exn /Wr]
openWr = /[String Exn /Wr]
```

`(openGenWr f create excl trunc p h)` opens the file `f` for writing. If `create` is true, and the file does not already exist, then `f` is created with permissions `p`. The given permissions are modified by the Unix process' umask in the normal way. If both `create` and `excl` are true then `openGenWr` fails if the specified file already exists. If `trunc` is true and the file does already exist, the file is truncated when it is opened. If `f` cannot be opened the handler `h` is called. `(openWr f h)` is a simplified version of `openGenWr`. If the file `f` already exists, it truncates it. If `f` does not already exist, it creates `f` with read and write permissions enabled. If `f` cannot be opened the handler `h` is called.

```
fnWr = /[/[String Sig] /Wr]
```

Builds a writer that calls the given function to process anything written on it.

3.14.4 Debugging Streams

```
traceRd = /[Rd String Wr /Rd]
traceWr = /[Wr String Wr /Wr]
```

`(traceRd rd s tr)` create a reader that behaves just like `rd` except that every operation on it is echoed to the tracing stream `tr` along with the identifying string `s`. `traceWr` behaves similarly, constructed a traced version of a writer.

3.15 Std/List: Lists

3.15.1 Types

```
#List : (Pos Type -> Type)
```

(List X) is the type of lists of elements of type X.

3.15.2 Creation

```
nil = (List Bot)
```

nil is the empty list.

```
cons = /[#X X (List X) /(List X)]
```

(cons x l) creates a new cons cell.

```
make = /[#X Int X /(List X)]
```

(make n x) creates a list of size n, with each cell containing x.

```
tabulate = /[#X Int /[Int /X] /(List X)]
```

(tabulate n f) creates a list of size n, such that the element with index i is initialised to (f i).

3.15.3 Interrogation

```
null = /[#X (List X) /Bool]
```

(null l) returns a boolean indicating whether l is the empty list.

```
car = /[#X (List X) /X]
unsafeCar = /[#X (List X) /X]
```

(car l) returns the head of the list l, if l is a cons cell. Causes a runtime failure if l is the empty list. unsafeCar behaves similarly, except that it does not test if l is the empty list.

```
cdr = /[#X (List X) /(List X)]
unsafeCdr = /[#X (List X) /(List X)]
```

(cdr l) returns the tail of the list l, if l is a cons cell. Causes a runtime failure if l is the empty list. unsafeCdr behaves similarly, except that it does not test if l is the empty list.

```
case = /[#X #R (List X) /[/R] /[X (List X) /R] /R]
```

(case l n c) returns the result of evaluating (n) if l is the empty list, otherwise it returns the result of evaluating (c hd tl) where hd and tl are the head and tail of the list l.

```
size = /[#X (List X) /Int]
```

(size l) returns the size of l.

```
nth = /[#X (List X) Int /X]
```

(nth l n) finds the nth element of the list l. Causes a runtime error if n << 0 or if n >= (size l).

3.15.4 Iteration

```
apply = /[#X (List X) /[X Sig] Sig]
revApply = /[#X (List X) /[X Sig] Sig]
```

`(apply l f)` applies `f` to each element of `l` (sequentially and in order). `revApply` behaves just like `apply`, except that it traverses `l` in reverse order.

```
itApply = /[#X (List X) /[Int X Sig] Sig]
revItApply = /[#X (List X) /[Int X Sig] Sig]
```

`(itApply l f)` applies `f` to each element of `l` (sequentially and in order), passing `f` the index of each element. `revItApply` behaves just the same as `itApply`, except that it traverses `l` in reverse order.

```
fold = /[#X #R (List X) /[X R /R] R /R]
revFold = /[#X #R (List X) /[X R /R] R /R]
```

`(fold l f i)` applies `f` to each element of `l` (sequentially and in order), passing `f` an accumulated result of type `R`. The initial accumulated result is `i`. `revFold` behaves just like `fold`, except that it traverses `l` in reverse order.

```
itFold = /[#X #R (List X) /[Int X R /R] R /R]
revItFold = /[#X #R (List X) /[Int X R /R] R /R]
```

`(itFold l f i)` applies `f` to each element of `l` (sequentially and in order), passing `f` the index of each element, and an accumulated result of type `R`. The initial accumulated result is `i`. `revItFold` behaves just like `itFold`, except that it traverses `l` in reverse order.

```
map = /[#X #R (List X) /[X /R] /(List R)]
revMap = /[#X #R (List X) /[X /R] /(List R)]
```

`(map l f)` maps the function `f` over each element in `l` (sequentially and in order). `revMap` behaves just like `map`, except that it traverses `l` in reverse order.

```
filter = /[#X (List X) /[X /Bool] /(List X)]
```

`(filter l f)` returns a list containing those elements of `l` for which `f` returns `true`.

3.15.5 Combination

```
rev = /[#X (List X) /(List X)]
```

`(rev l)` reverses the list `l`.

```
append = /[#X (List X) (List X) /(List X)]
```

`(append l1 l2)` appends the lists `l1` and `l2`.

```
revAppend = /[#X (List X) (List X) /(List X)]
```

`(revAppend l1 l2)` reverses `l1` and appends it to `l2`.

3.15.6 Sorting

```
sort = /[#X (List X) /[X X /Cmp] Bool /(List X)]
```

`(sort l f d)` sorts `l` according to the comparison function `f`. If `d` is `true`, then all but one of each set of elements of `l` that are judged equal by `f` will be dropped.

3.15.7 Conversion

```
hash = /[#X (List X) /[X /Int] /Int]
```

Given a hash function `f` for values of type `X`, `(hash l f)` returns a hash value for a list `l` of type `(List X)`.

```
toStream = /[#X (List X) /(Stream X)]
```

`(toStream l)` converts the list `l` into a stream (cf. Section 3.25).

3.15.8 Comparison

```
cmp = /[#X (List X) (List X) /[X X /Cmp] /Cmp]
```

Given an comparison function `f` for values of type `X`, `(cmp l1 l2 f)` returns a single value indicating the ordering of `l1` and `l2` (cf. Section 3.6).

3.15.9 Printing

```
pr = /[#X (List X) String String String /[X Sig] Sig]
```

`(pr l beg sep end prx)` prints the list `l`. It first prints `beg`, then prints all the elements of `l` (using the supplied printing function `prx`, separating each element with `sep`) and finishes by printing `end`.

3.15.10 Abbreviations

```
val nil  = list.nil
val cons = list.cons
val null = list.null
val car  = list.car
val cdr  = list.cdr
```

3.16 `Std/MD5`: MD5 Message Digests

The module contains various functions for calculating MD5 message-digests. The code was derived from the RSA Data Security, Inc. MD5 message-digest algorithm (see original RSA copyright in Section 3.16.5).

3.16.1 Types

```
#MD5Context
```

An `MD5Context` represents the intermediate state of a MD5 message-digest calculation.

```
#MD5Digest
```

An `MD5Digest` is a 128-bit message-digest (or checksum).

3.16.2 Operations

```
init = /[/MD5Context]
```

`(init)` creates an initial `MD5Context`.

```
update = /[MD5Context Bytes idx=Int len=Int Sig]
```

`(update c b idx=i len=l)` updates the MD5 context `c` with `l` characters from the byte array `b`, starting at index `i`.

```
updateString = /[MD5Context String idx=Int len=Int Sig]
```

(`updateString c s idx=i len=l`) updates the MD5 context `c` with `l` characters from the string `s`, starting at index `i`.

```
finalise = /[MD5Context /MD5Digest]
```

(`finalise c`) signals that we have finished adding data to the context `c`, and computes the final message-digest.

3.16.3 Comparison

```
eq = /[MD5Digest MD5Digest /Bool]
cmp = /[MD5Digest MD5Digest /Cmp]
```

Equality and comparison functions for MD5 message-digests.

3.16.4 Conversion

```
toString = /[MD5Digest /String]
```

(`toString d`) converts the message-digest `d` to a 32-character hex string.

```
td = (TD MD5Digest)
```

A type descriptor for MD5 message-digests.

```
fromString = /[String /MD5Digest]
```

(`fromString s`) calculates the message-digest for the single string `s`.

```
fromBytes = /[Bytes /MD5Digest]
```

(`fromBytes b`) calculates the message-digest for the single byte array `b`.

3.16.5 RSA Copyright

©1991-2, RSA Data Security, Inc. Created 1991. All rights reserved. License to copy and use this software is granted provided that it is identified as the "RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing this software or this function. License is also granted to make and use derivative works provided that such works are identified as "derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided "as is" without express or implied warranty of any kind. These notices must be retained in any copies of any part of this documentation and/or software.

3.17 `Std/Match`: Wildcard Matching

This module provides a simple facility for matching strings against wildcard patterns.

3.17.1 Types

```
#Match
```

`Match` is the type of a compiled pattern match expression.

3.17.2 Compiling and Matching

```
compile = /[String Exn /Match]
```

(compile s h) compiles a regular expression s into a form that is suitable for subsequent matches. The error handler h is called if s does not denote a valid wildcard pattern.

```
match = /[Match String Int Int /Bool]
```

(search m s i len) returns true if m matches the substring of s starting at index i and of length len. It generates a runtime error if i and len do not denote a valid substring of s.

3.18 Std/Misc: Miscellaneous Utilities

3.18.1 Types

```
#Sig = /[]
```

Sig is a convenient abbreviation for output channels which carry the empty tuple. Such channels are used, by convention, to signal various conditions, such as the completion of operations.

3.18.2 Tuple Operations

```
fst = /[#X [X] /X]
snd = /[#X #Y [X Y] /Y]
thd = /[#X #Y #Z [X Y Z] /Z]
```

fst, snd and thd are functions for projecting the first, second and third components of a tuple. Note that record subtyping allows these functions to be applied to longer tuples. For example, the expression (snd [3 4 5]) is well-typed and evaluates to 4.

3.18.3 Channel Input/Output

```
chan = /[#X /^X]
```

The expression (chan) returns a fresh channel.

```
read = /[#X ?X /X]
```

The expression (read c) returns a value which has been read from c.

```
write = /[#X !X X Sig]
```

The expression (write c v) asynchronously writes the value v on c.

```
forward = /[#X ?X /X]
```

The expression (forward c r) repeatedly reads values from c, and forwards them to the responsive channel r.

```
rchan = /[#X !X //X]
```

The expression (rchan r) creates and returns a responsive channel and forwards values from it to the ordinary channel r.

3.18.4 Discarding Results

```
discard = /[#X //X]
```

(discard) returns a process abstraction that accepts a value of type X and throws it away. Useful for calling functions that expect result channels when the result is actually not needed: for example, writing int.pr![5 (discard)] creates a process that prints 5 on the standard output, but does not do anything special when it finishes.

```
await = /[#X X /[]]
```

The expression (await v) evaluates v and then returns [].

3.18.5 Function Composition

```
compose = /[#X #Y #Z /[Y /Z] /[X /Y] //[X /Z]]
```

(compose g f) composes the functions g and f.

3.18.6 Printing

The following printing functions just use the standard buffered input/output operations provided by ANSI C. They do not provide asynchronous I/O. If you want proper non-blocking I/O then use the facilities provided in the Io module (Section 3.14).

```
pr = /[String Sig]
```

(pr s) prints the string s on stdout.

```
prNL = /[String Sig]
```

(prNL s) prints the string s, followed by a newline, on stdout.

```
nl = /[Sig]
```

(nl) prints a newline on stdout.

```
prErr = /[String Sig]
```

(prErr s) prints the string s on stderr.

```
prErrNL = /[String Sig]
```

(prErrNL s) prints the string s, followed by a newline, on stderr.

```
nlErr = /[Sig]
```

(nlErr) prints a newline on stderr.

```
flush = /[Sig]
```

(flush) flushes all buffered output.

```
print = /String
printi = /Int
```

print!s prints the string s on stdout asynchronously. printi!i prints the integer i on stdout asynchronously. (These primitives are used mainly for simple examples in the tutorial. In real programs, pr is better.)

3.18.7 Abbreviations

```
val fst = misc.fst
val snd = misc.snd
val thd = misc.thd
val chan = misc.chan
val read = misc.read
val write = misc.write
val forward = misc.forward
val rchan = misc.rchan
val discard = misc.discard
val await = misc.await
val compose = misc.compose
val pr = misc.pr
val prNL = misc.prNL
val nl = misc.nl
val prErr = misc.prErr
val prErrNL = misc.prErrNL
val nlErr = misc.nlErr
val print = misc.print
val printi = misc.printi
```

3.19 `Std/Opt`: Optional Values

This type is used by various functions to optionally return values.

3.19.1 Types

```
#Opt : (Pos Type -> Type)
```

The type `(Opt X)` represents an optional value of type `X`.

3.19.2 Construction

```
yes = /[#X X /(Opt X)]
no = (Opt Bot)
```

`(yes x)` converts a value `x` into an optional value. `no` is the null optional value.

3.19.3 Operations

```
case = /[#X #R (Opt X) /[/R] /[X /R] /R]
```

`(case o absent present)` returns the value computed by `(absent)` if `o` is the null optional value. Otherwise, it returns the value returned by `(present x)` where `x` is the value contained in `o`.

```
get = /[#X (Opt X) X /X]
```

`(get o d)` returns the contents of `o`, if present. It returns the default value `d` if `o` is the null value.

```
map = /[#X #Y /[X /Y] (Opt X) /(Opt Y)]
```

`(map o f)` applies the function `f` to the contents of `o` (if present).

3.19.4 Printing

```
pr = /[#X (Opt X) String /[X Sig] Sig]
```

`(pr o no yes)` prints out the optional value `o`. If `o` is not present, we print the string `no`, otherwise we print the contents of `o` using the supplied printing function `yes`.

3.20 `Std/PP`: Pretty Printing

A simple pretty-printing abstraction.

3.20.1 Types

```
#PP =
  [format = /[Wr Int Int /Int]
   sizes  = (List Int)
   breaks = Int]
```

A pretty printer is an object with the following fields: `format` takes as arguments an output stream, upon which it writes the formatted output, and two integers which record the required number of breaks and the current column number respectively. It returns the number of characters that it consumed. The `sizes` field is a (non-empty) list of integers which indicate the number of columns which the pretty printer needs to print itself. The `i`'th element of the list gives the size if `i` boxes are broken. The `breaks` field indicates how many times the pretty printer can break its output so that it consumes fewer columns.

3.20.2 Construction

```
null = PP
```

The `null` value is a pretty printer which does nothing. It has size of zero, and cannot be broken into smaller components.

```
string = /[String /PP]
```

`(string s)` returns a pretty printer which prints out `s` on a single line. The size of the pretty printer is equal to the size of the string `s`. `(string s)` cannot be broken into smaller components. The string `s` must not contain any control characters, such as newline, tab or linefeed.

```
int = /[Int /PP]
```

`(int i)` returns a pretty printer which prints out `i`. The size of `(int i)` is equal to the length of the string representation of `i`. `(int i)` cannot be broken into smaller components.

```
bool = /[Bool /PP]
```

`(bool b)` returns a pretty printer which prints out `b`. The size of `(bool b)` is equal to the length of the string representation of `b`. `(bool b)` cannot be broken into smaller components.

3.20.3 Operations

```
extend = /[Int Char Int Char PP /PP]
```

`(extend left lc right rc pp)` extends the pretty printer `pp` by inserting `left` copies of the character `lc` to the left of `pp` and `right` copies of the character `rc` to the right of `pp`.

```
break = /[Int PP /PP]
```

`(break n pp)` forces `pp` to break at least `n` times.

```
leftJustify = /[Int PP /PP]
```

`(leftJustify width pp)` finds the widest possible size of `pp` which is less than or equal to `width`, and returns a pretty printer which formats `pp` at that size, appending as many spaces after `pp` as are required. If `pp` cannot fit into `width` we generate a runtime error.

```
rightJustify = /[Int PP /PP]
```

`(rightJustify width pp)` finds the widest possible size of `pp` which is less than or equal to `width`, and returns a pretty printer which formats `pp` at that size, appending as many spaces before `pp` as are required. If `pp` cannot fit into `width` we generate a runtime error.

```
center = /[Int PP /PP]
```

`(center pp)` fixes the size of `pp` to be `width`. If `pp` cannot fit into `width` we generate a runtime error.

```
hbox = /[(List PP) /PP]
vbox = /[(List PP) /PP]
```

`(hbox l)` combines the pretty printers in `l` horizontally. The resulting pretty printer will always print `l` out horizontally. `(vbox l)` behaves similarly, except that it combines the pretty printers in `l` vertically.

```
hvbox = /[(List PP) /PP]
```

`(hvbox l)` combines the pretty printers in `l` either vertically or horizontally, depending on whether it is asked to break or not.

```
list = /[(List PP) start=PP before=PP after=PP finish=PP /(List PP)]
```

`(list start=s before=b after=a finish=f l)` returns a new list of pretty printers which starts with `s`, finishes with `f` and where `b` is inserted before each element of `l` and `a` is inserted after each element of `l`.

3.21 `Std/Prim`: Primitive Operations

This module defines various primitives which are required to build the standard libraries (basic arithmetic and string operations, for instance). There is no need for ordinary programs to use the operations defined here, since they are re-exported from other user-level modules. These primitives must be defined here to avoid creating cyclic dependencies between modules such as `Std/Int` and `Std/String`.

3.21.1 Types

```
#CString
```

The type of C strings (Pict strings have an explicit length files, unlike C strings which are null-terminated).

```
#Word
```

Pict currently used tagging to help with garbage collection. One problem with this is that the type `Int` is not the same size as a machine word. A value of type `Word` represents a full-width machine word (and is currently heap-allocated).

3.21.2 Operations

```
+ = /[Int Int /Int]
```

Integer addition.

```
== = /[Int Int /Bool]
<> = /[Int Int /Bool]
>> = /[Int Int /Bool]
<< = /[Int Int /Bool]
>= = /[Int Int /Bool]
<= = /[Int Int /Bool]
```

Ordering functions on integers.

```
size = /[String /Int]
```

(size s) returns the size of the string s.

```
+$ = /[String String /String]
```

(+$ s t) concatenates s and t

```
==$ = /[String String /Bool]
```

(==$ s t) returns true if s and t are equal.

```
fromCString = /[CString /String]
```

Converts a C string into a Pict string.

```
shortenTo = /[String Int /[]]
```

(shorten s x) sets the length of s to x. Thus, whenever s is garbage collected it will be shortened.

```
shorten = /[String /[]]
```

(shorten s) counts the number of characters in s before the first null character, and then sets the length of s to match. Thus, whenever s is garbage collected it will be shortened.

3.21.3 Manipulating Bytes

```
alloc = /[Int /String]
```

(alloc n) allocates n bytes of memory.

3.21.4 Manipulating Words

```
allocWord = /[/Word]
```

(word) allocates enough memory to store a single machine word. This function is only intended for use with inline C code.

```
readWord = /[Word /Int]
writeWord = /[Word Int /[]]
```

Read and write integer values into a word.

3.21.5 Testing Pointers

```
nullPtr = /[#X X /Bool]
```

(nullPtr x) checks if x is a null pointer.

3.22 Std/Queue: Queues

Queues are implemented as so that that insertion and deletion only use a small, constant, amount of time and space (as long as the queue is used in a FIFO or LIFO manner).

3.22.1 Types

```
#Queue : (Pos Type -> Type)
```

The type of functional queues.

3.22.2 Creation

```
empty = (Queue Bot)
```

`empty` is the empty queue.

```
make = /[#X Int X /(Queue X)]
```

`(make n x)` creates a queue of size `n`, with each element containing `x`.

```
tabulate = /[#X Int /[Int /X] /(Queue X)]
```

`(tabulate n f)` creates a queue of size `n`, such that the element with index `i` is initialised to `(f i)`.

3.22.3 Interrogation

```
isEmpty = /[#X (Queue X) /Bool]
```

`(isEmpty q)` tests whether `q` is empty.

```
size = /[#X (Queue X) /Int]
```

`(size q)` returns the size of `q`.

3.22.4 Modification

```
insertHd = /[#X X (Queue X) /(Queue X)]
insertTl = /[#X X (Queue X) /(Queue X)]
```

Insert an element at the head or tail of a queue.

```
removeHd = /[#X (Queue X) /[X (Queue X)]]
removeTl = /[#X (Queue X) /[X (Queue X)]]
```

Remove an element from the hear or tail of a queue, returning a pair of the removed element and the rest of the queue. Generates a fatal error if the queue is empty.

3.22.5 Iteration

```
apply = /[#X (Queue X) /[X Sig] Sig]
revApply = /[#X (Queue X) /[X Sig] Sig]
```

`(apply q f)` applies `f` to each element of `q` (sequentially and in order). `(revApply f q)` behaves similarly, except that it traverses `q` in reverse order.

```
itApply = /[#X (Queue X) /[Int X Sig] Sig]
revItApply = /[#X (Queue X) /[Int X Sig] Sig]
```

`(itApply q f)` applies `f` to each element of `q` (sequentially and in order), passing `f` the index of each element. `(revItApply f q)` behaves similarly, except that it traverses `q` in reverse order.

```
fold = /[#X #R (Queue X) /[X R /R] R /R]
revFold = /[#X #R (Queue X) /[X R /R] R /R]
```

(fold q f) applies f to each element of q (sequentially and in order), passing f an accumulated result of type R. The initial accumulated result is init. (revFold q f) behaves similarly, except that it traverses l in reverse order.

```
itFold = /[#X #R (Queue X) /[Int X R /R] R /R]
revItFold = /[#X #R (Queue X) /[Int X R /R] R /R]
```

(itFold q f) applies f to each element of q (sequentially and in order), passing f the index of each element, and an accumulated result of type R. The initial accumulated result is init. (revItFold q f) behaves similarly, except that it traverses q in reverse order.

```
map = /[#X #R (Queue X) /[X /R] /(Queue R)]
revMap = /[#X #R (Queue X) /[X /R] /(Queue R)]
```

(map q f) maps f over each element in q (sequentially and in order). (revMap q f) behaves similarly, except that it traverses q in reverse order.

3.22.6 Conversion

```
toStream = /[#X (Queue X) /(Stream X)]
```

(toStream q) converts the queue q into a stream.

```
hash = /[#X (Queue X) /[X /Int] /Int]
```

Given a hash function h for values of type X, (hash q h) returns a hash value for q.

3.22.7 Combination

```
rev = /[#X (Queue X) /(Queue X)]
```

(rev q) reverses the queue q.

```
append = /[#X (Queue X) (Queue X) /(Queue X) ]
```

(append q1 q2) appends the queues q1 and q2.

3.22.8 Comparison

```
cmp = /[#X (Queue X) (Queue X) /[X X /Cmp] /Cmp]
```

Given an comparison function f for values of type X, (cmp q1 q2 f) returns a single value indicating the ordering of q1 and q2 (cf. Section 3.6).

3.22.9 Printing

```
pr = /[#X (Queue X) String String String /[X Sig] Sig]
```

(pr q beg sep end prx) Prints the queue q. It first prints beg, then prints all the elements of q (using the supplied printing function prx, separating each element with sep) and finishes by printing end.

3.23 `Std/Random`: Random Numbers

The random number generator used in this library is the PMMMLCG (Prime Modulus M Multiplicative Linear Congruential Generator), a modified version of the random number generator proposed by Park and Miller in "Random Number Generators: Good Ones Are Hard to Find", CACM October 1988, Vol 31, No. 10. It includes modifications proposed by Park to provide better statistical properties (i.e. more 'random' – less correlation between sets of generated numbers). It was developed by John Burton (`jcburt@cs.wm.edu`) of G & A Technical Software, Inc 28 Research Drive Hampton, Va. 23666.

The generator is of the form $x = (x * A)\%M$ and has a period of $2^{31} - 1$, with numbers generated in the range of $0 < x < M$. The generator can run on any machine with a 32-bit integer, without overflow. The choice of A and M can radically modify the properties of the generator. The current values (48271 and 2147483647) were chosen after followup work to the original paper mentioned above.

3.23.1 Operations

```
randomize = /[Int /[]]
```

`(randomize x)` sets the seed in the random number generator to `x`.

```
max = Int
```

The upper limit on numbers produces by `random`.

```
random = /[/Int]
```

Returns a random integer `r` such that `0` < `r` < `max`. Note that this function never returns zero, unlike `randomInt`.

```
randomInt = /[Int /Int]
```

`(randomInt m)` returns a random integer `r` such that `0` ≤ `r` < `m`. Note that this function may return zero, unlike `random`.

```
randomCoin = /[/Bool]
```

A special case of the random number generator that returns a random boolean.

3.24 `Std/Ref`: Reference Cells

3.24.1 Types

```
#Ref = \X =
  [get = /[/X]
   put = /[X Sig]]
```

The type `(Ref X)` represents a reference cell, and supports `get` and `put` methods for reading and writing to the cell.

3.24.2 Creation

```
make = /[#X X /(Ref X)]
```

`(make i)` creates a reference cell with initial value `i`.

```
empty = /[#X /(Ref X)]
```

`(empty)` creates a reference cell with no initial value. Calls to the `get` method of an empty reference cell will block until a value is supplied using the `put` method.

3.25 `Std/Stream`: Streams

3.25.1 Types

```
#Stream = \X = ![/[] /X]
```

We can interrogate a stream of type `(Stream X)` by sending it a pair of channels `[e r]`. The stream will signal on `e` if it is empty, or return a value of type `X` along `r` if it is non-empty.

3.25.2 Operations

```
isEmpty = /[#X (Stream X) /Bool]
```

`(isEmpty s)` tests whether the stream `s` is empty.

```
get = /[#X (Stream X) /X]
```

`(get s)` gets the next element of the stream `s` (generating a runtime error if `s` is empty).

```
append = /[#X (Stream X) (Stream X) /(Stream X)]
```

`(append s1 s2)` appends the streams `s1` and `s2`.

3.25.3 Comparison

```
cmp = /[#X (Stream X) (Stream X) /[X X /Cmp] /Cmp]
```

`(cmp s1 s2 f)` returns a single value indicating the ordering of `s1` and `s2`.

3.26 `Std/String`: Strings

3.26.1 Types

The type `String` is built in and cannot be redefined.

3.26.2 Creation

```
make = /[Char Int /String]
```

`(make c n)` creates a string of size `n`, containing the character `c`.

```
tabulate = /[Int /[Int /Char] /String]
```

`(tabulate n f)` creates a string of size `n`, such that the character with index `i` is initialised to `(f i)`.

```
+$ = /[String String /String]
```

`(+$ s t)` concatenates `s` and `t`.

3.26.3 Interrogation

```
size = /[String /Int]
```

`(size s)` returns the number of characters in `s`.

```
nth = /[String Int /Char]
unsafeNth = /[String Int /Char]
```

`(nth s n)` returns the `n`'th character of `s`. If `n` is not in the range $0 \leq$ `n` $<$ `(size s)` then we generate a runtime error. `unsafeNth` behaves just like `nth`, except that it does not do any range checking.

```
sub = /[String Int Int /String]
```

Extract the substring of `s`, starting at `start` and of length `len`. It must be the case that `len` ≥ 0, `start` ≥ 0 and `start` $+$ `len` $\leq$ `(size s)`.

3.26.4 Comparison

```
cmp = /[String String /Cmp]
```

Returns a single value indicating the ordering of `s` and `t` (cf. Section 3.6).

```
==$ = /[String String /Bool]
<>$ = /[String String /Bool]
```

`s` equal to `t`, and `s` not equal to `t`.

```
>>$ = /[String String /Bool]
<<$ = /[String String /Bool]
```

`s` strictly greater than `t`, and `s` strictly less than `t`.

```
>=$ = /[String String /Bool]
<=$ = /[String String /Bool]
```

`s` greater than or equal to `t`, and `s` less than or equal to `t`.

```
in = /[String String /Bool]
```

`(in s t)` returns `true` whenever `s` occurs as a substring of `t`.

```
prefix = /[String String /Bool]
```

`(prefix s t)` returns `true` whenever `s` occurs as a prefix of `t`.

```
indexIn = /[Char String Exn /Int]
```

`(indexIn c s h)` returns the index of the first occurrence of `c` in `s`, calling the error handler `h` if `c` does not occur in `s`.

```
occursIn = /[Char String /Bool]
```

`(occursIn c s)` returns `true` if the character `c` occurs in the string `s`.

3.26.5 Conversion

```
hash = /[String /Int]
```

`(hash s)` returns a hash value for `s`.

```
split = /[String String /(List String)]
```

`(split sep s)` splits `s` into a list of tokens. Any character in the string `sep` is considered to be a separator. For example, `(split " \n" "a b\n c")` returns the tokens `"a"`, `"b"` and `"c"`.

```
toString = /[String /String]
```

`(toString s)` converts all non-printable characters in `s` into printable characters using Pict's character escape sequences. Newlines are converted to `\n`, `"` is converted to `\"` and `\` is converted to `\\`, all other non-printable characters are converted to a character literal of the form `\ddd` where `d` is a decimal digit.

```
fromList = /[(List Char) /String]
```

`(fromList l)` converts the list of characters `l` to a string.

3.26.6 Printing

```
pr = /[String Sig]
```

(pr s) prints the string s on stdout.

3.26.7 Abbreviations

```
val +$ = string.+$
val ==$ = string.==$
val <>$ = string.<>$
val >>$ = string.>>$
val <<$ = string.<<$
val >=$ = string.>=$
val <=$ = string.<=$
```

3.27 Std/Tags: Unique Tags

3.27.1 Types

```
#Tag
```

A value of type Tag can be used as a unique identifier for an object.

3.27.2 Creation

```
make = /[/Tag]
```

(make) creates a globally unique tag.

```
dummy = Tag
```

A dummy tag (which is guaranteed to be distinct from all tags generated using make).

3.27.3 Comparison

```
<< = /[Tag Tag /Bool]
<= = /[Tag Tag /Bool]
== = /[Tag Tag /Bool]
<> = /[Tag Tag /Bool]
>= = /[Tag Tag /Bool]
>> = /[Tag Tag /Bool]
```

The above functions provide an ordering on tags.

```
cmp = /[Tag Tag /Cmp]
```

(cmp t1 t2) returns a single value indicating the ordering of t1 and t2 (cf. Section 3.6).

3.27.4 Conversion and Printing

```
hash = /[Tag /Int]
```

(hash t) Returns a hash value for t.

```
toString = /[Tag /String]
```

(toString t) converts t to a string.

```
pr = /[Tag Sig]
```

(pr t) prints t on stdout.

Chapter 4

Posix Libraries

4.1 `Posix/Args`: Command-line Arguments

This module provides some simple functions for determining the arguments used to invoke the current Unix process, as well as a more sophisticated argument parsing interface.

4.1.1 Types

```
#Spec = [String String]
```

A value of type `Spec` specifies a command-line option. The first string is used to indicate the full name name of the option. The second string may indicate an abbreviation for the full name (or may be the same as the full name if no abbreviated version is required).

4.1.2 Simple Argument Interrogation

```
argc = Int
```

The number of arguments used to invoke the current Unix process.

```
argv = /[Int /String]
```

`(argv n)` returns the `n`'th argument used to invoke the current Unix process. If $\texttt{n} < \texttt{0}$ or $\texttt{n} \geq \texttt{argc}$ then we generate a runtime error.

4.1.3 Argument Parsing Functions

```
getListOpt =
  /[(List Spec) (List Spec) (List Spec) (List String) Exn
     /[/[String Exn /Int]
       /[String Exn /String]
       /[String /Bool]
       (List String)]]
```

`(getListOpt intSpecs strSpecs unitSpecs args h)` parses the argument list `args` according to the given option specifications, calling the error handler `h` if an illegal option is encountered. We parse three different kinds of options: integer options of the form `-name <integer>`, string options of the form `-name <string>` and nullary options of the form `-name`. We return three functions which can be used to ask which options were specified in the argument list `args`. `getListOpt` also returns a list of those arguments which were not consumed during option parsing (we call such arguments *anonymous arguments*).

Parsing proceeds as follows:

1. If the current argument is `-`, we add `-` to the list of anonymous arguments (since `-` is used, by convention, to indicate the standard input or standard output streams).
2. If the current argument is `--`, we stop parsing and add all the remaining arguments (except the `--`) to the list of anonymous arguments.
3. Otherwise, if the current argument has the form `-name`, we lookup `name` in the specified lists of integer, string and unitary options.
   - If `name` matches either the full name or the abbreviated name of an integer option, we check that at least one more argument exists (and that it denotes a valid integer). If there are no more arguments, or the next argument does not denote a valid integer, we signal an error on the given error handler.
   - If `name` matches a string option, we just check that at least one more argument exists. If there are no more arguments, we signal an error on the given error handler.
   - If `name` matches a nullary option, we simply record that the option was present.
   - If `name` does not match any of the specified options, we signal an error on the given error handler.
4. If the current argument does not start with a `-`, we add it to the list of anonymous arguments.

Anonymous arguments are returned in the same order in which they were encountered by the parser. The three functions returned by `getListOpt` can be used to find out whether a given option was specified in the argument list `args`. Note that a given option name (or abbreviated option name) must not be defined more than once in the lists `intSpecs`, `strSpecs` and `unitSpecs`.

```
getOpts =
  /[(List Spec) (List Spec) (List Spec) Exn
     /[/[String Exn /Int]
       /[String Exn /String]
       /[String /Bool]
       (List String)]]
```

`(getOpts intSpecs strSpecs unitSpecs h)` parses the arguments used to invoke the current Unix process according to the specification given in `intSpecs`, `strSpecs` and `unitSpecs`. Calls the error handler `h` if the arguments could not be parsed correctly. The parsing algorithm is the same as that used in `getListOpts`.

4.2 `Posix/Env`: Environment Variables

This module provides some simple functions for examining the environment variable bindings for the current Unix process.

```
get = /[String Exn /String]
```

`(get n h)` searches the environment list for a string that matches the name `n`, calling the given error handler if no match could be found.

```
toList = /[/(List String)]
```

`(toList)` returns a list of all environment entries. Each entry is of the format `NAME=value`. The order in which strings appear in the environment is not significant, but the same `NAME` must not appear more than once.

4.3 `Posix/Fd`: Descriptor-based Input/Output

This module implements a procedural interface to Posix's descriptor-based input/output. Some care is required when using these operations, since many have the potential to block the whole Pict system. This module is not intended for general use: we provide high-level non-blocking file objects in the `Io` module (Section 3.14).

4.3.1 Types

```
#Fd
```

Values of type `Fd` represent Unix file descriptors.

```
#AccessMode
```

File access modes specify whether a file descriptor to be used for reading, writing, or both.

4.3.2 Standard Input and Output

```
stdIn = Fd
stdOut = Fd
stdErr = Fd
```

The values `stdIn`, `stdOut` and `stdErr` correspond to the standard Unix file descriptors of the same name.

4.3.3 Reading and Writing

The following operations read and write data on file descriptors. If the file descriptor being written to or read from is set to be asynchronous, and the read/write operation would block, we suspend the operation until the next `SIGIO` and then retry.

```
read = /[Fd Bytes Int Int /Int]
```

`(read fd b i n)` reads a maximum of `n` characters into `b`, starting at offset `i`, and returns the number of characters read.

```
write = /[Fd String Sig]
```

`(write fd s)` writes the string `s` on `fd`.

```
writeChar = /[Fd Char Sig]
```

`(writeChar fd c)` writes the character `c` on `fd`.

```
writeSubstring = /[Fd String Int Int Sig]
```

`(writeSubstring fd s i n)` writes `n` characters beginning from position `i` in the string `s` (it must be the case that `i` and `n` denote a valid substring of `s`).

```
writeBytes = /[Fd Bytes Int Int Sig]
```

`(writeBytes fd b i n)` writes `n` characters from `b` beginning from position `i` (it must be the case that that `i` and `n` denote a valid sub-array of `b`).

4.3.4 Access Modes

```
readOnly = AccessMode
writeOnly = AccessMode
readWrite = AccessMode
```

File access modes specify whether a file descriptor to be used for reading, writing, or both. The access modes are chosen when the file descriptor is opened, and never change.

4.3.5 Opening and Closing Files

```
open = /[String AccessMode Bool Bool Bool Perm Exn /Fd]
```

`(open f a create excl truncate p h)` opens the file `f` using the access mode `a`. If `create` is true, and the file does not already exist, then `f` is created with permissions `p` (the given permissions are modified by the Unix process' umask in the normal way). If both `create` and `excl` are true then `open` fails if the specified file already exists. If `truncate` is true and `f` already exists, `f` is truncated when it is opened. Calls the error handler `h` if `f` cannot be opened.

```
openRd = /[String Exn /Fd]
```

`(openRd f h)` is a simplified version of `open` which opens the file `f` for reading, calling the error handler `h` if the file cannot be opened.

```
openWr = /[String Exn /Fd]
```

`(openWr f h)` is a simplified version of `open` which opens the file `f` for writing. If `f` already exists, it truncates it. If `f` does not already exist, it creates `f` with read and write permissions enabled.

```
close = /[Fd Sig]
```

`(close fd)` closes `fd`. Invoking any operations on `fd` after it has been closed causes a runtime error.

4.3.6 Misc

```
mkASync = /[Fd Exn Sig]
```

`(mkASync fd h)` tries to make the file descriptor `fd` use asynchronous I/O. It calls the error handler `h` if it fails to make `fd` asynchronous. Note that all the read operations provided in this module work correctly with asynchronous file descriptors (if no data is available to be read, they block only the caller, not the whole runtime system).

4.4 `Posix/File`: File and Directory Manipulation

This module provides some simple functions for manipulating directories and files.

4.4.1 Types

```
#Dir
```

The type `Dir` represents a directory as a stream of directory entries.

4.4.2 Setting File Permissions

```
chmod = /[String Perm Exn Sig]
```

`(chmod f p h)` sets the permissions of the file `f` to `p`, calling the error handler `h` if the permissions could not be changed.

4.4.3 Removing Files

```
unlink = /[String Exn Sig]
```

`(unlink f h)` deletes the file `f`, calling the error handler `h` if `f` could not be deleted. The filename `f` must not denote a directory.

```
rmdir = /[String Exn Sig]
```

`(rmdir d h)` deletes the directory `d`, calling the error handler `h` if `d` could not be deleted. The directory `d` must be empty.

4.4.4 Renaming Files

```
rename = /[String String Exn Sig]
```

(`rename f n h`) renames the file `f` to `n`, calling the error handler `h` if any problems are encountered.

4.4.5 Hard Links

```
link = /[String String Exn Sig]
```

(`link f l h`) creates a new (hard) link named `l` to the existing file `f`, calling the error handler `h` if any problems are encountered.

4.4.6 Working Directory

```
getcwd = /[/String]
```

(`getcwd`) returns the working directory of the current unix process.

4.4.7 Creating Directories

```
mkdir = /[String Perm Exn Sig]
```

(`mkdir d p h`) creates the directory `d` with the permissions `p`, calling the error handler `h` if the directory could not be created.

4.4.8 Directory Streams

```
openDir = /[String Exn /Dir]
```

(`openDir d h`) opens the directory `d`, calling the error handler `h` if `d` is not a directory, or if `d` could not be opened.

```
closeDir = /[Dir Sig]
```

(`closeDir dir`) closes the directory stream `dir`.

```
readDir = /[Dir Exn /String]
```

(`readDir dir h`) reads the next directory entry from the directory `dir`. Note that we never return entries for `"."` or `".."`, even though these are always valid file names in any Unix directory.

4.4.9 Iterating over Directories

```
applyDir = /[String Exn /[String Sig] Sig]
```

(`applyDir d h f`) applies the function `f` to every entry in the directory `d`, calling the error handler `h` if `d` could not be opened.

```
foldDir = /[#X String Exn /[String X /X] X /X]
```

(`foldDir d h f i`) applies the function `f` to every entry in the directory `d`, accumulating a value of type `X`. Calls the error handler `h` if `d` could not be opened.

```
listDir = /[String Exn /(List String)]
```

(`listDir d h`) returns the contents of the directory `d` as a list of strings, calling the error handler `h` if `d` could not be opened.

4.4.10 Lock Files

```
lockFile = /[String Exn Sig]
```

(`lockFile f h`) creates the lock file `f`. This is an atomic operation. Calls the given error handler if the lock file could not be created (this could either be because some other process has already created the lock file or because of other filesystem problems).

```
unlockFile = /[String Exn Sig]
```

(`unlockFile f h`) deletes the lock file `f`, calling the given error handler if the lock file could not be removed.

4.5 `Posix/Perm`: File Permissions

4.5.1 Types

```
#AccessPerm =
  [read    = Bool
   write   = Bool
   execute = Bool]
```

The type `AccessPerm` has a boolean field for each different kind of access permission that a user can have for a file (or directory): `read` indicates if we can read the file (for directories, this means we can list the contents of the directory); `write` indicates if we can write to the file (for directories, this means we can create and remove files in the directory); and `execute` indicates whether we can run the file as a program (for directories, this means we can access files in the directory).

```
#Perm =
  [setuid = Bool
   setgid = Bool
   sticky = Bool
   user   = AccessPerm
   group  = AccessPerm
   other  = AccessPerm]
```

A value of type `Perm` records access permissions for three categories of users: `user` holds the access permissions for the user who owns the file; `group` records the access permissions for the users who are in the file's group; and `other` gives the access permissions for everyone else.

In addition, `Perm` records three special attributes, which affect only executable files and, on some systems, directories: the `setuid` bit signals if the process' effective user id is set to that of the file upon execution (has no effect on directories); the `setgid` bit signals if the process' effective group id is set to that of the file upon execution (on some systems, if this bit is set on a directory, this forces files created in the directory to be put into the same group as the directory, no matter what group the user who creates them is in); and the `sticky` bit signals that the program's text image should be saved on the swap device so it will load more quickly when run (on some systems, if this bit is set on a directory, it prevents users from removing files that they do not own in the directory).

4.5.2 Construction

```
___ = AccessPerm
__x = AccessPerm
_w_ = AccessPerm
_wx = AccessPerm
r__ = AccessPerm
r_x = AccessPerm
rw_ = AccessPerm
rwx = AccessPerm
```

Abbreviations for all eight kinds of access permissions, named in the same way as access permissions are named by unix utilities. For example, `rw_` allows reading and writing but not execution.

```
null = Perm
```

The null permission (all permission attributes are set to `false`).

4.5.3 Combination

```
neg = /[Perm /Perm]
```

`(neg p)` negates the permission bits in `p`.

```
add = /[Perm Perm /Perm]
```

`(add p1 p2)` combines the permissions in `p1` and `p2` by computing the boolean disjunction of corresponding permission bits in `p1` and `p2`.

```
del = /[Perm Perm /Perm]
```

`(del p1 p2)` deletes from `p1` those permissions which are set in `p2` by computing the boolean exclusive-or of corresponding permission bits in `p1` and `p2`.

```
mask = /[Perm Perm /Perm]
```

`(mask p1 p2)` masks `p1` by `p2` by computing the boolean conjunction of corresponding permission bits in `p1` and `p2`.

4.5.4 Conversion

```
toInt = /[Perm /Int]
```

`(toInt p)` converts the permission bits in `p` into an integer which is suitable for passing to Posix C functions (this functionality should not be required by user programs).

```
fromInt = /[Int /Perm]
```

`(fromInt i)` performs the inverse of `toInt`, converting the integer `i` into a permission value.

4.5.5 Printing

```
pr = /[Perm Sig]
```

`(pr p)` prints the permission bits `p` in the same format as is used by `ls`.

4.6 `Posix/Persona`: Process Persona

The priveleges assigned to a Unix process are a function of its *persona*. A process' persona is determined by its *effective user id*, *effective group id*, and a set of *supplementary group ids*. In normal circumstances, all child processes inherit their persona from their parents. However, it is possible in some circumstances to modify the effective user and group ids of a process. Some programs can be executed with different user and group ids to those of the parent process. In this case, the *real user id* and *real group id* identify the user who created the process, but the effective user and group ids may be different.

4.6.1 Types

```
#UID<Int
#GID<Int
```

UID and GID are the types of user and group identifiers, respectively.

```
#PasswordInfo =
  [userName = String
   realName = String
   password = String
   user     = UID
   group    = GID
   homeDir  = String
   shell    = String]
```

A record of type PasswordInfo contains various information about a user, and is stored in the system password database.

4.6.2 User Identifiers

```
uid = /[/UID]
```

Returns the real user id of the current Unix process.

```
euid = /[/UID]
```

Returns the effective user id of the current Unix process.

4.6.3 Group Identifiers

```
gid = /[/GID]
```

Returns the real group id of the current Unix process.

```
egid = /[/GID]
```

Returns the effective group id of the current Unix process.

```
groups = /[/(List GID)]
```

Returns a list of the supplementary groups of the current Unix process.

4.6.4 Password Database

```
uidInfo = /[UID Exn /PasswordInfo]
```

(uidInfo uid h) returns the information about uid in the system password database. Calls the handler h if there is no entry for uid in the database.

```
userInfo = /[String Exn /PasswordInfo]
```

(userInfo user h) returns the information about user in the system password database. Calls the handler h if there is no entry for user in the database.

4.7 Posix/Proc: Process Creation

This module provides some simple facilities for running Unix processes.

4.7.1 Types

```
#PID
```

`PID` is the type of proceess identifiers.

```
#Status
```

`Status` is the type of process status information.

4.7.2 Process Identification

```
getpid = /[/PID]
```

Returns the process id of the current process.

```
getppid = /[/PID]
```

Returns the process id of the parent of the current process.

```
pidString = /[PID /String]
```

`(pidString p)` Converts the process id `p` to a string.

4.7.3 Process Status

```
waitpid = /[PID /Status]
```

`(wait p)` waits for the child process `p` to terminate, returning the exit status of `p`.

```
exited = /[Status /Bool]
```

`(exited s)` returns `true` if the child process terminated normally with `exit` or `_exit`.

```
exitStatus = /[Status /Int]
```

If `(exited s)` is true then `(exitStatus s)` returns the low-order 8 bits of the exit status value from the child process.

```
signalled = /[Status /Bool]
```

Returns `true` if the child process terminated because it received a signal that was not handled.

```
termSig = /[Status /Signal]
```

If `(signalled s)` is true then `(termSig s)` returns the signal number of the signal that terminated the child process.

4.7.4 Process Creation

```
exec = /[String (List String) /PID]
```

`(exec prog args)` executes the program `prog` with the given arguments (by creating a separate Unix process). Returns the process id of the created process.

```
exece = /[String (List String) (List String) /PID]
```

`exece` is similar to `exec` except that it allows an explicit environment (a list of strings of the form `VAR=value`, see Section 4.2 for more details) to be passed to the program `prog`. Returns the process id of the created process.

```
shell = /[String /Status]
```

`(exec com)` executes the shell command `com` (using `/bin/sh`) and then waits for it to terminate.

```
shelle = /[String (List String) /Status]
```

A variant of `shell` which allows an explicit environment to be supplied for the shell process (a list of strings of the form `NAME=value`).

4.8 `Posix/Signals`: Posix Signals

This library implements a low-level interface to Unix signals.

4.8.1 Types

```
#Signal
```

The type of Posix signals.

```
#SignalResult = [wokeUp=Bool stayInstalled=Bool persistent=Bool]
#SignalHandler = /[last=Bool gotSignal=Bool /SignalResult]
```

A `SignalHandler` is a process which is woken up when a signal arrives, or when there are no other processes left to be executed by the runtime system.

A signal handler takes two boolean arguments: the first indicates whether any other processes are currently executing, the second indicates whether we have actually received a signal. If there are no other processes running and we have not received any signals, then it is likely that the runtime system is just about to suspend itself. A useful thing that a signal handler can do at such a time is to flush its output buffers. For example, in our widgets library the signal handler which controls the connection to the X server flushes all pending drawing operations before allowing the runtime system to suspend itself (otherwise updates to windows controlled by Pict processes might be delayed indefinately).

Each signal handler returns a record of type `SignalResult`. The `wokeUp` field indicates whether the signal handler woke up any processes (this tells the runtime system that it should not suspend itself, since some new processes have been created, presumably as the result of receiving some signal). The `stayInstalled` field indicates whether the handler wishes to remain installed for the next signal. The runtime system uses `persistent` field to calculate whether (in the case where there are no runnable processes) it should terminate the whole runtime system. If any signal handlers indicate that they are persistent, then we take that as an indication that some Pict processes have some real work to do (once some signal arrives). Note that each signal handler is queried just before we suspend/terminate the runtime system. This enables each handler to examine its own internal data structures to decide whether they currently contain any user-level Pict processes which are waiting for a signal.

4.8.2 Signals

```
SIGINT = Signal
SIGQUIT = Signal
SIGALRM = Signal
SIGCONT = Signal
SIGCHLD = Signal
SIGHUP = Signal
SIGIO = Signal
SIGTERM = Signal
SIGVTALRM = Signal
```

There are a number of different signals which can be handled from Pict. See your Posix documentation for the meaning of individual signals.

4.8.3 Registering Handlers

```
register = /[Signal SignalHandler Sig]
```

`(register s h)` registers the handler `h` so that it is invoked when the next signal `s` arrives. There is no limit on the number of handlers which can be registered for a given signal (all relevant handlers are invoked when the signal arrives).

```
retry = /[#X Signal /[/(Opt X)] /X]
```

A simplified interface to `register`. `retry` is given a signal and a body, which it executes. The body returns either `opt.no`, indicating that it did not succeed in performing its job and needs to block until the signal `s` is received and then try again, or else `(opt.yes x)` indicating that the the body completed successfully and `x` should be returned as the result of the whole `retry`. If `s` is received while the body is still executing and the body then returns `opt.no`, the body will be started again immediately.

4.9 `Posix/Sockets`: Socket Utilities

This library implements a low-level interface to the Posix sockets facility.

4.9.1 Types

```
#Port = Int
```

Communication ports are integers in the range `0` to `65535` (note that we don't currently do any range checking on port numbers).

```
#Address
```

The type `Address` represents a pair of a hostname and a port number.

```
#Socket < Fd
```

Sockets are a special kind of file descriptor.

4.9.2 Service Ports

```
reservedPort = Port
userReservedPort = Port
```

Standard servers (such as `ftp` and `telnet`) use port numbers less than `reservedPort` (as we will see below, you can look up such services by name). If you write a server that is not one of the standard ones, you should use a number greater than `userReservedPort` since such numbers will never be generated automatically by the system. Unfortunately, no facility is available to avoid conflicts with servers being run by other users.

```
getServicePortByName = /[String Exn /Port]
```

Looks up the service port associated with `name` in the database of standard system services. Calls the given error handler if no matching service is found.

4.9.3 Host Addresses

```
getHostByName = /[String Port Exn /Address]
```

`(getHostByName name p h)` looks up `name` is the hosts database. If successful, it pairs the given port number `p` with the resulting address and returns an address suitable for use with `bind` and `connect`. If no host called `name` can be found, it calls the error handler `h`.

```
addressString = /[Address /String]
```

Converts an address to a string.

4.9.4 Socket Creation

```
streamSocket = /[Exn /Socket]
```

We only provide support for sockets which use the Internet protocol. `(streamSocket h)` creates a socket which has sequenced, reliable, two-way connection-based byte streams, calling the error handler `h` if a socket could not be created.

4.9.5 Accepting Connections

```
bind = /[Socket Address Int Exn Sig]
```

(`bind s a i h`) binds the socket `s` to the address `a` and then enables client connections (`i` specifies how many pending client connections are allowed before we start rejecting new connections). Calls the error handler `h` if anything fails.

```
bindAny = /[Socket Port Int Exn Sig]
```

(`bindAny s p i h`) binds the socket `s` to the port `p` on the local machine and then enables client connections (`i` specifies how many pending client connections are allowed before we start rejecting new connections). Calls the error handler `h` if anything fails.

```
accept = /[Socket Exn /Fd]
```

(`accept s h`) waits for a client to make a connection to the socket `s`, returning a new file descriptor which can be used to read and write to the client. The socket `s` must already be bound to an address (using one of the above functions). If `s` is set to be asynchronous and if there are no connections pending then `accept` will suspend itself until the next `SIGIO` and then retry the accept operation.

4.9.6 Making Connections

```
connect = /[Socket Address Exn /Fd]
```

(`connect s a h`) makes a connection between `s` and the socket at the address `a` (the socket addressed by `a` must already be set up as a server). The error handler `h` is called if a connection could not be made. Upon successful connection to `a`, we return a file descriptor which can be used to read and write to the server at `a`. If `s` is set to be asynchronous and connecting to `a` would block then `connect` will suspend itself until the next `SIGIO` and then retry the connection operation.

4.10 `Posix/Stat`: File Status Queries

A simple interface to the Posix `stat` function. We currently do not make all the information that is returned by `stat` accesible to Pict programs.

4.10.1 Types

```
#FileType < Int
```

A value of type `FileType` tells us what kind of file.

```
#Time = Int
```

An abbreviation for integers used to represent system times (in seconds since January 1st, 1970).

```
#Stat =
  [dev = FileType     {- Regular, directory, symlink, or other -}
   perm = Perm        {- Permission bits -}
   size = Int         {- Size in bytes -}
   atime = Time       {- Unix atime -}
   mtime = Time       {- Unix mtime -}
   ctime = Time]      {- Unix ctime -}
```

The type of results of file status queries.

4.10.2 Construction

```
regular = FileType
directory = FileType
symlink = FileType
other = FileType
```

Constructors for the four recognized unix file types.

4.10.3 Conversion

```
prFileType = /[FileType Sig]
```

`(prFileType ft)` prints the file type `ft`.

```
fileTypeTD = (TD FileType)
```

Type descriptor for file types.

4.10.4 File Status Queries

```
stat = /[String Exn /Stat]
```

Low-level interface to the Posix `stat` system call.

```
fileExists = /[String /Bool]
```

`(fileExists f)` returns `true` if the file `f` exists.

4.11 `Posix/Time`: Time and Date

4.11.1 Types

```
#Time =
  [seconds = Int    {- 0..59 -}
   minutes = Int    {- 0..59 -}
   hours = Int      {- 0..23 -}
   days = Int       {- 1..31 -}
   months = Int     {- 1..12 -}
   years = Int      {- Years since 0 AD -}
   weekDays = Int   {- Days since Sunday (0..6) -}
   yearDays = Int]  {- Days since January 1 (0..365) -}
```

4.11.2 Operations

```
localTime  = /[/Time]
```

Gets the current local time.

```
gmTime  = /[/Time]
```

Gets the current time represented in Greenwich mean time.

4.11.3 Conversion

```
weekDayString  = /[Int /String]
```

(weekDayString d) returns a three-character string (Mon, Tue, Wed, ...) for each day of the week (generates a runtime error if d is not in the range 0 to 6).

```
monthString  = /[Int /String]
```

(monthString m) returns a three-character string (Jan, Feb, Mar, ...) for each month of the year (generate a runtime error if m is not in the range 1 to 12).

```
toString  = /[Time /String]
```

(toString t) converts a time t into a string with the following fields, each separated by a single space: weekDay, month, date, hours:minutes:seconds, year.

4.11.4 Printing

```
pr = /[Time Sig]
```

(pr t) prints out the time t using the same format as toString.

Chapter 5

Contributed Libraries

5.1 `Contrib/Chun`: Network Channels

This library implements a physically distributed channel abstraction, using Unix sockets as the underlying transport layer. This abstraction can be used to build Pict programs that operate “semi-transparently” across a physical network. (The original version of this library was written by Cédric Fournet and Benjamin Pierce.)

Type descriptors (Section 5.5) are used for packing and unpacking values. Any value for which a type descriptor can be constructed may be transmitted across the network. Such types include integers, strings, tuples, recursively defined types, and user-defined types. This library use Internet sockets to implement type descriptors for output-only channels and responsive channels. This enables output-only channels and responsive channels to be transparently communicated across the network to other Pict programs.

5.1.1 Types

```
#Chun =
  [register = /[#X String (TD X) /X Sig]
   lookup   = /[#X (TD X) String Port String //X]
   rchanTD  = /[#X (TD X) /(TD /X)]
   ochanTD  = /[#X (TD X) /(TD !X)]]
```

`(register n tdx c)` associates a globally visible name `n` with the local channel `c`, allowing Pict processes on remote machines to connect to it using `lookup` (the type descriptor `tdx` is used to unmarshall data sent across the network to `c`).

`(lookup tdx h p n)` contacts the server at host `h` and port `p` and asks to be connected to the channel named `n`. `lookup` returns a responsive channel of type `/X`, which can be used to send values of type `X` to the remote machine `h` (the type descriptor given when we called `lookup` is used to marshall any data sent to the remote machine).

`rchanTD` and `ochanTD` are implementations of type descriptors for responsive channels and output-only channels respectively (these operations are defined here, since they only make sense once we have started a server running on our local machine).

5.1.2 Starting a Server

```
start = /[Port Exn /Chun]
```

`(start p h)` starts a server on the local machine which enables Pict processes to receive data from remote machines (`p` specifies which port the server should listen on for incoming data). The server calls the error handler `h` if it encounters any errors.

5.2 Contrib/Clock: A Clock Server

This package provides a simple clock server. Processes can await a single clock tick or awakened on every clock tick. The clock can be started, stopped, and the period of clock ticks may be changed. A process can ask to be notified when the clock either stops or starts.

5.2.1 Waiting for Clock Ticks

```
awaitTick = /[Sig]
```

The expression `(awaitTick)` blocks the current process until the next clock tick.

```
everyTick = /Sig
```

The output `everyTick!c` causes the clock to send a message along `c` for every clock tick.

5.2.2 Starting and Stopping the Clock

```
stop = /[Sig]
go = /[Sig]
```

`(stop)` stops the clock (if the clock has already stopped, it does nothing). `(go)` starts the clock (if the clock is already running, it does does nothing).

```
onStop = /Sig
onGo = /Sig
```

`onStop!c` and `onGo!c` cause the clock to send a message along `c` whenever the clock is stopped or started, respectively. Messages are only sent when a real change in the clock status occurs, and not for every call of the `stop` and `go` methods.

```
toggle = /[Sig]
```

`(toggle)` restarts the clock, if it was stopped, or stops the clock if it was already running.

```
tick = /[Sig]
```

`(tick)` manually ticks the clock, regardless of whether the clock is running or not.

```
getDelay = /[/Int]
setDelay = /[Int Sig]
```

`(getDelay)` gets the current delay between clock ticks (in hundredth's of a second). `(setDelay x)` sets the delay between clock ticks to `x`. The delay `x` must be non-negative, or we generate a runtime error. A value of `0` makes the clock tick as fast as possible (100 times a second).

5.3 Contrib/Parse: Parsing Utilities

```
int = /[Rd Exn /Int]
```

`(int s h)` parses an integer (a non-empty sequence of digits, possibly prefixed by the negation symbol `~`) from `s`. Calls the handler `h` if it cannot parse an integer from `s`.

```
while = /[Rd /[Char /Bool] /String]
```

`(while s f)` collects all those characters in `s` up to (but not including) the next character for which `f` returns false.

```
line = /[Rd /String]
```

`(line s)` returns those characters in `s` up to (but not including) the next newline character.

```
whitespace = /[Rd Sig]
```

`(whitespace s)` removes any leading whitespace (spaces, newlines or tabs) from `s`.

```
match = /[Rd String Exn Sig]
```

`(match s str)` extracts a string of length `(string.size str)` from `s` and checks that it matches `str`, calling the given handler if the string parsed from the stream fails to match `str`.

```
char = /[Rd Char Exn Sig]
```

`(char s ch)` extracts the next character from `s` and checks that it matches `ch`, calling the given handler if the character fails to match (or there was no character available).

5.4 `Contrib/Ppm`: Portable Pixmap Utilities

The portable pixmap (ppm) format is a generic file format for color bitmaps, supported by a number of publically available utilities. This library allows ppm files to be loaded and printed from Pict.

```
parse = /[#X Rd Exn /[Int Int Int /X] /(Array2 X)]
```

`(parsePpm s h f)` given a function `f` which maps rgb triples to values of type `X`, `parse` reads a ppm from `s` and converts it into an array of `X`'s. It calls the error handler `h` if it cannot find a valid ppm.

```
print = /[#X Wr String (Array2 X) /[X /[Int Int Int]] Bool Sig]
```

`(print s name p f bin)` is the inverse of `parse`. Given a conversion function from values of type `X` to rgb triples, it writes a ppm on `s`. There are two kind of ppm file: the `bin` flag indicates whether we should use the (more efficient) binary format. The `name` argument is stored in the ppm file.

5.5 `Contrib/TD`: Type Descriptors

Type descriptors support typesafe conversions between typed Pict values and streams of characters. Applications include storing persistent data structures in files); marshalling data to be transmitted from one address space to another; and implementing a simple `Dynamic` type and associated `typecase` construct.

Not every Pict type has an associated type descriptor. In particular, there is no way to build a type descriptor for a channel type. This is because the `TD` library is entirely written in Pict, with no “cheating,” and there is no typesafe way to coerce between strings and channels. Moreover, type descriptors are not generated automatically. However, this library does provides predefined descriptors for some common base types and facilities for building type descriptors for more complex types by composing type descriptors.

5.5.1 Types

```
#TD = \X =
  [marshall = /[Wr X Sig]
   unmarshall = /[Rd Exn /X]]
```

`(TD X)` denotes a type descriptor for a type `X` and is a record containing two functions: `(marshall w x)` prints a string representation of `x` on `w`, while `(unmarshall r h)`, parses an element of `X` from `r`, signalling on `h` if we fail to parse a value of type `X`.

5.5.2 Descriptors For Basic Types

Type descriptors for basic types.

```
int = (TD Int)
char = (TD Char)
bool = (TD Bool)
string = (TD String)
bytes = (TD Bytes)
list = /[#X (TD X) /(TD (List X))]
```

5.5.3 Descriptors For Abstract Datatypes

```
adt = /[#ADT #Rep /[ADT /Rep] /[Rep /ADT] (TD Rep) /(TD ADT)]
```

`(adt project inject repTD)` allows type descriptors to be built for arbitrary abstract datatypes. The functions `project` and `inject` are used to convert values between the abstract type `ADT` and the representation type `Rep`. To marshall a value of type `ADT`, we first convert it to a value of type `Rep`, using the `project` function, and then marshall that value using `repTD`, a type descriptor for the type `Rep`. To unmarshall a value of type `ADT` we first unmarshall a value of type `Rep` using `repTD`. We then use `inject` to convert the resulting value to a value of type `ADT`.

5.5.4 Descriptors For Tuples

Given type descriptors for each of the component types, these functions build type descriptors for tuple types of various widths.

```
unit = (TD [])
singleton = /[#X (TD X) /(TD [X])]
pair = /[#X #Y (TD X) (TD Y) /(TD [X Y])]
triple = /[#X #Y #Z (TD X) (TD Y) (TD Z) /(TD [X Y Z])]
quad = /[#W #X #Y #Z (TD W) (TD X) (TD Y) (TD Z) /(TD [W X Y Z])]
```

5.5.5 Descriptors For Recursive Types

```
recursive =
  /[#M:(Type->Type) /[(TD (rec X = (M X))) /(TD (M (rec X = (M X))))]
    /(TD (rec X = (M X)))]
```

Given a type `T` of the form `(rec X = (M X))`, we can build a type descriptor for `T` by supplying `recursive` with a function from `(TD T)` to `(TD (M T))` (i.e a type descriptor for the "skeleton" `M` that is parametric in a type descriptor for `T` itself).

Chapter 6

X11 Libraries

6.1 `X11/Xlib`: X Library Interface

This library implements a low-level interface to the `Xlib` X library interface. We only give brief descriptions of the types and functions defined here. Please refer to the `Xlib` documentation for more detailed information [SG92, Nye90][1].

6.1.1 Types

```
#Display
```

The type of display connections.

```
#ScreenNum
```

Each display supports a number of screens. Each screen is referenced by a `ScreenNum`.

```
#XID
```

Values of type `XID` identify many different kinds of resources (such as fonts, windows and pixmaps).

```
#XContext : (Type -> Type)
```

An value of type `(XContext X)` is a table which maps resource identifiers of type `XID` to values of type `X`.

```
#Xfunction
```

Values of type `Xfunction` are used when you update a section of a drawable (the destination) with bits from somewhere else (the source). An `Xfunction` value defines how the new destination bits are to be computed from the source bits and the old destination bits.

```
#Color
```

The type `Color` corresponds to what `Xlib` calls an `XColor` and contains both RGB values and a pixel value (which can be used for drawing operations).

```
#Font
```

The `Font` type defined here actually corresponds to what `Xlib` calls a `FontStruct`, since it contains extra information about the font (enabling us to, for instance, calculate text extents).

[1] Some of our documentation uses extracts from the X Consortium `Xlib` manual pages

```
#TextExtents =
  [fontAscent = Int
   fontDescent = Int
   lbearing = Int
   rbearing = Int
   width = Int
   ascent = Int
   descent = Int]
```

Information about a particular font.

```
#GC
```

A `GC` contains various pieces of style information for drawing functions (such as foreground and background color, font, etc.).

```
#Drawable < XID
```

The type `Drawable` is a subtype of `XID`, and denotes those resource identifiers which can sensibly be the target of drawing operations.

```
#Pixmap < Drawable
```

The type `Pixmap` is a subtype of `Drawable` and denotes an off-screen area on the server which can be the target of any drawing operation.

```
#Window < Drawable
```

The type `Window` is a subtype of `Drawable` and denotes an on-screen area on the server which can be the target of any drawing operation. Unlike a `Pixmap`, the client program is responsible for maintaining the contents of a `Window`.

```
#Geometry =
  [x = Int
   y = Int
   width = Int
   height = Int
   borderWidth = Int
   depth = Int]
```

The type `Geometry` describes the size and position on a `Drawable`, along with the width of its border and the depth of the display.

```
#KeySym < Int
```

The type `KeySym` is used to return the symbolic name of a key that has been pressed.

```
#Button
```

The type of mouse buttons.

```
#XEvent
```

The type `XEvent` can be though of as the union of a number of specific event types.

```
#EventMask
```

An `EventMask` can be used to indicate to the X server which events we are interested in receiving.

6.1.2 Displays

```
openDisplay = /[String Exn /Display]
```

(`openDisplay name h`) Opens a connection to the named X server. If `name` is the empty string, we use the value of the `DISPLAY` environment variable to determine the name of the X server. Calls the error handler `h` if the server connection could not be made.

```
closeDisplay = /[Display Sig]
```

(`closeDisplay d`) closes the specified connection to the display `d`. No resources allocated by the client on the display `d` should be referenced after calling `closeDisplay`, since it deallocates all resources created on `d`. It is good to call `closeDisplay` explicitly before exiting so that any pending errors are reported.

```
displayFd = /[Display /Fd]
```

(`displayFd d`) returns the underlying file descriptor used to communicate with the X server. This is necessary to implement non-blocking I/O on the X server connection.

```
synchronize = /[Display Bool Sig]
```

(`synchronize d b`) makes interaction with the display `d` synchronous, if `b` is `true`, or asynchronous if `b` is `false` (useful for debugging, since errors will be reported synchronously).

```
flush = /[Display Sig]
```

(`flush d`) flushes the output buffer to the display `d`. Most applications need not use this function because the output buffer is automatically flushed when we check for events. However, it may occasionally be advantageous to call `flush` to avoid flicker when updating a window.

6.1.3 Screens

```
defaultScreen = /[Display /ScreenNum]
```

(`defaultScreen d`) returns the default screen for the display `d`.

```
defaultDepth = /[Display ScreenNum /Int]
```

(`defaultDepth d s`) returns the default depth of the screen `s` on the display `d`.

6.1.4 Resource Identifiers

```
uniqueContext = /[#X /(XContext X)]
```

(`uniqueContext`) allocates a fresh table, suitable for storing values of type `X`.

```
findContext = /[#X Display XID (XContext X) Exn /X]
```

(`findContext d xid c h`) looks up the value associated with `xid` in the context `c`, calling the error handler `h` if there is no mapping for `xid`.

```
saveContext = /[#X Display XID (XContext X) X Sig]
```

(`saveContext d xid c x`) associates the value `x` with `xid` in the context `c`, removing the previous entry for `xid` (if one exists).

```
deleteContext = /[#X Display XID (XContext X) Exn Sig]
```

(`deleteContext d xid c h`) removes the value associated with `xid` from the context `c`, calling the error handler `h` if there is no mapping for `xid`.

6.1.5 Drawing Styles

```
gXclear = Xfunction          {- 0                        -}
gXand = Xfunction            {- src AND dst              -}
gXandReverse = Xfunction     {- src AND NOT dst          -}
gXcopy = Xfunction           {- src                      -}
gXandInverted = Xfunction    {- (NOT src) AND dst        -}
gXnoop = Xfunction           {- dst                      -}
gXxor = Xfunction            {- src XOR dst              -}
gXor = Xfunction             {- src OR dst               -}
gXnor = Xfunction            {- (NOT src) AND (NOT dst) -}
gXequiv = Xfunction          {- (NOT src) XOR dst        -}
gXinvert = Xfunction         {- NOT dst                  -}
gXorReverse = Xfunction      {- src OR (NOT dst)         -}
gXcopyInverted = Xfunction   {- NOT src                  -}
gXorInverted = Xfunction     {- (NOT src) OR dst         -}
gXnand = Xfunction           {- (NOT src) OR (NOT dst)  -}
gXset = Xfunction            {- 1                        -}
```

`gXcopy` is typically the most useful drawing attribute because it will work on a color display, but special applications may use other functions, particularly in concert with particular planes of a color display.

6.1.6 Colors

```
colorRed = /[Color /Int]
colorGreen = /[Color /Int]
colorBlue = /[Color /Int]
```

`(colorRed c)`, `(colorGreen c)` and `(colorBlue c)` return the red, green and blue intensities of the color `c`. (All RGB intensities are integers in the range `0` to `65535`.)

```
allocColor = /[Display ScreenNum Int Int Int (Handler [Int Int Int]) /Color]
```

`(allocColor d s r g b h)` allocates a read-only colormap entry (in the default colormap of the screen `s`) corresponding to the closest RGB value supported by the hardware. Calls the given error handler if the color could not be allocated. Multiple clients that request the same effective RGB value can be assigned the same read-only entry, thus allowing entries to be shared. When the last client deallocates a shared cell, it is deallocated.

```
allocNamedColor = /[Display ScreenNum String Exn /Color]
```

`(allocNamedColor d s n h)` looks up and allocates the color `n` in the default colormap of the screen `s`. If `n` is not in the Host Portable Character Encoding, the result is implementation dependent. Calls the error handler `h` if the color could not be allocated.

6.1.7 Fonts

```
loadQueryFont = /[Display String Exn /Font]
```

`(loadQueryFont d n h)` loads the font named `n`. If `n` is not in the Host Portable Character Encoding, the result is implementation dependent. If the font does not exist, we call the error handler `h`.

```
freeFont = /[Display Font Sig]
```

`(freeFont f)` explicitly deallocates `f` (`f` should not be subsequently referenced).

```
textExtents = /[Font String /TextExtents]
```

`(textExtents f s)` find the extents of the string `s` when rendered using the font `f`.

6.1.8 Graphics Contexts

```
defaultGC = /[Display ScreenNum /GC]
```

(`defaultGC d s`) gets the default `GC` for the screen `s` on the display `d`.

```
setBackground = /[Display GC Color Sig]
```

(`setBackground d gc c`) sets the background color attribute in `gc` to `c`.

```
setForeground = /[Display GC Color Sig]
```

(`setForeground d gc c`) sets the foreground color attribute in `gc` to `c`.

```
setFunction = /[Display GC Xfunction Sig]
```

(`setFunction d gc f`) sets the drawing function attribute in `gc` to `f`.

```
setFont = /[Display GC Font Sig]
```

(`setFont d gc f`) sets the font attribute in `gc` to `f`.

6.1.9 Drawables

```
getGeometry = /[Display Drawable /Geometry]
```

(`getGeometry d dr`) returns the geometry of the drawable `dr`.

6.1.10 Drawing Operations

```
copyArea = /[Display Drawable Drawable GC Int Int Int Int Int Int Sig]
```

(`copyArea d src dest gc sx sy sw sh dx dy`) combines the specified rectangle of `src` with the specified rectangle of `dest`. The drawables must have the same root and depth. If regions of the source rectangle are obscured and have not been retained in backing store or if regions outside the boundaries of the source drawable are specified, those regions are not copied. Instead, the following occurs on all corresponding destination regions that are either visible or are retained in backing store. If the destination is a window with a background other than None, corresponding regions of the destination are tiled with that background (with plane-mask of all ones and `gXcopy` function). Regardless of tiling or whether the destination is a window or a pixmap, if graphics-exposures is `true`, then `GraphicsExpose` events for all corresponding destination regions are generated. If graphics-exposures is `true` but no `GraphicsExpose` events are generated, a `NoExpose` event is generated. Note that by default graphics-exposures is `true` in new graphics contexts.

```
copyPlane = /[Display Drawable Drawable GC Int Int Int Int Int Int Int Sig]
```

(`copyPlane d src dest gc sx sy sw sh dx dy plane`) uses a single bit plane of the specified source rectangle combined with the specified `GC` to modify the specified rectangle of `dest`. The drawables must have the same root but need not have the same depth. `plane` must have exactly one bit set to `1` and have a values less than 2^n where where n is the depth of `src`.

```
drawArc = /[Display Drawable GC Int Int Int Int Int Int Sig]
fillArc = /[Display Drawable GC Int Int Int Int Int Int Sig]
```

(`drawArc d dr gc x y w h a1 a2`) draws a single circular or elliptical arc, specified by a rectangle and two angles. The center of the circle or ellipse is the center of the rectangle, and the major and minor axes are specified by the width and height. `a1` specifies the start of the arc relative to the three-o'clock position from the center, in units of $degrees * 64$. `a2` specifies the path and extent of the arc relative to the start of the arc, in units of $degrees * 64$. Positive angles indicate counter clockwise motion, and negative angles indicate clockwise motion. If the magnitude of `a2` is greater than `360` degrees, `xDrawArc` truncates it to `360` degrees. `fillArc` is similar to `drawArc`, except that it fills the area inside the arc.

```
drawImageString = /[Display Drawable GC Int Int String Sig]
```

(`drawImageString d dr gc x y s`) fills a destination rectangle with the background pixel defined in `gc`, and then draws the string `s` at position `[x y]` using the foreground pixel and font attributes defined in `gc`. The upper-left corner of the filled rectangle is at `[x (y-ascent)]`, the width is `width`, the height is `ascent+descent`, where `width`, `ascent`, and `descent` are the same as would be returned by `textExtents`.

```
drawLine = /[Display Drawable GC Int Int Int Int Sig]
```

(`drawLine d dr gc x1 y1 x2 y2`) draws a line between the points `[x1 y1]` and `[x2 y2]`.

```
drawPoint = /[Display Drawable GC Int Int Sig]
```

(`drawPoint x y`) draws a point at the position `[x y]`.

```
drawQuad = /[Display Drawable GC Int Int Int Int Int Int Int Int Sig]
fillQuad = /[Display Drawable GC Int Int Int Int Int Int Int Int Sig]
```

(`drawQuad x1 y1 x2 y2 x3 y3 x4 y4`) draws a closed polygon with vertices `[x1 y1]`, `[x2 y2]`, `[x3 y3]` and `[x4 y4]`. `fillQuad` is similar to `drawQuad`, except it fills the area inside the polygon.

```
drawRectangle = /[Display Drawable GC Int Int Int Int Sig]
fillRectangle = /[Display Drawable GC Int Int Int Int Sig]
```

(`drawRectangle d dr gc x y w h`) draws the outline of a rectangle, connecting the following four points: `[x y]`, `[(x+w) y]`, `[(x+w) (y+h)]`, and `[x (y+h)]`. `fillRectangle` is similar to `drawRectangle`, except it fills the area inside the rectangle.

```
drawString = /[Display Drawable GC Int Int String Sig]
```

(`drawString d dr gc x y s`) draws the string `s` at position `[x y]` using the foreground pixel and font attributes defined in `gc`. Each character image, as defined by the font in `gc`, is treated as an additional mask for a fill operation on the drawable. The drawable is modified only where the font character has a bit set to `1`.

```
drawTriangle = /[Display Drawable GC Int Int Int Int Int Int Sig]
fillTriangle = /[Display Drawable GC Int Int Int Int Int Int Sig]
```

(`drawTriangle d dr gc x1 y1 x2 y2 x3 y3`) draws a triangle with vertices `[x1 y1]`, `[x2 y2]` and `[x3 y3]`. `fillTriangle` is similar to `drawTriangle`, except it fills the area inside the triangle.

6.1.11 Pixmaps

```
createPixmap = /[Display Drawable Int Int Int /Pixmap]
```

(`createPixmap d dr w h depth`) creates a pixmap of width, height, and depth you specified and returns a pixmap ID that identifies it. The width and height arguments must be non-zero. The depth argument must be one of the depths supported by the screen of the drawable `dr`. The server uses `dr` to determine on which screen to create the pixmap. The pixmap can be used only on this screen and only with other drawables of the same depth (see `copyPlane` for an exception to this rule). The initial contents of the pixmap are undefined.

```
freePixmap = /[Display Pixmap Sig]
```

(`freePixmap d p`) deallocates the pixmap `p`. The pixmap should never be referenced again.

6.1.12 Windows

```
createWindow = /[Display Window Int Int Int Int Int Color Color /Window]
```

`(createWindow d win x y w h borderWidth bg border)` creates an unmapped, input-output subwindow for a parent window `win`, returning the window ID of the created window. The created window is placed on top in the stacking order with respect to siblings. Any part of the window that extends outside its parent window is clipped. The created window inherits its depth, class, and visual from its parent. All other window attributes, except background and border (which are specified explicitly), have their default values.

```
defaultRootWindow = /[Display /Window]
```

`(defaultRootWindow d)` returns the default root window for the display `d`.

```
destroyWindow = /[Display Window Sig]
```

`(destroyWindow d win)` destroys the window `win` and all of its subwindows. The window should never be referenced again. If `win` is mapped, it is unmapped automatically.

```
mapWindow = /[Display Window Sig]
```

`(mapWindow d win)` maps the window `win` and all of its subwindows that have had map requests. Mapping a window that has an unmapped ancestor does not display the window but marks it as eligible for display when the ancestor becomes mapped. This function has no effect if the window is already mapped.

```
unmapWindow = /[Display Window Sig]
```

`(unmapWindow d win)` unmaps the window `win`. If the specified window is already unmapped, `unmapWindow` has no effect. Any child window of `win` will be invisible until `win` is mapped again.

```
moveWindow = /[Display Window Int Int Sig]
```

`(moveWindow d win x y)` moves the window `win` to the specified `x` and `y` coordinates, but does not change the window's size, raise the window, or change the mapping state of the window. Moving a mapped window may or may not lose the window's contents depending on if the window is obscured and if no backing store exists. If the contents of the window are lost, the X server generates `Expose` events.

```
resizeWindow = /[Display Window Int Int Sig]
```

`(resizeWindow d win w h)` changes the inside dimensions of the window `win`, not including its borders. This function does not change the window's upper-left coordinate or the origin and does not restack the window. Changing the size of a mapped window may lose its contents and generate `Expose` events.

```
moveResizeWindow = /[Display Window Int Int Int Int Sig]
```

`(moveResizeWindow d win x y w h)` changes both the size and location of the window `win`.

```
clearWindow = /[Display Window Sig]
```

`(clearWindow d win)` clears the entire window `win` (without generating any `Exposure` events).

```
clearArea = /[Display Window Int Int Int Int Bool Sig]
```

`(clearArea d w x y w h e)` paints a rectangular area in the window `win` according to the specified dimensions using the window's background pixel or pixmap. If `w` is zero, it is replaced with the current width of the window minus `x`. If `h` is zero, it is replaced with the current height of the window minus `y`. If the window has a defined background tile, the rectangle clipped by any children is filled with this tile. If `e` is `true`, one or more `Expose` events are generated for regions of the rectangle that are either visible or are being retained in a backing store.

```
setWindowBackground = /[Display Window Color Sig]
```

(setWindowBackground d win c) sets the background color of the window win to c. Changing the background does not cause the window contents to be changed.

```
setWindowBackgroundPixmap = /[Display Window Pixmap Sig]
```

(setWindowBackgroundPixmap d win p) sets the background pixmap of the window win to p. The background pixmap can immediately be freed if no further explicit references to it are to be made.

```
setWindowBorder = /[Display Window Color Sig]
```

(setWindowBorder d win c) sets the border color of win to c.

```
setWindowBorderWidth = /[Display Window Int Sig]
```

(setWindowBorderWidth d win x) sets the border width of win to x.

```
storeName = /[Display Window String Sig]
```

(storeName d win s) assigns the name s to the window win. A window manager can display the window name in some prominent place, such as the title bar, to allow users to identify windows easily. Some window managers may display a window's name in the window's icon, although they are encouraged to use the window's icon name if one is provided by the application. If the string is not in the Host Portable Character Encoding, the result is implementation dependent.

```
withdrawWindow = /[Display Window ScreenNum /Bool]
```

(withdrawWindow d win s) unmaps the window win and sends a synthetic UnmapNotify event to the root window of the screen s. Window managers may elect to receive this message and may treat it as a request to change the window's state to withdrawn. When a window is in the withdrawn state, neither its normal nor its iconic representations are visible. Returns true if the UnmapNotify event was successfully sent, false otherwise.

6.1.13 KeySyms

```
XKHome  = KeySym
XKLeft  = KeySym
XKRight = KeySym
XKUp    = KeySym
XKDown  = KeySym
```

We only provide definitions for a few of common keysyms; others can be added very straightforwardly.

6.1.14 Buttons

```
button1 = Button
button2 = Button
button3 = Button
button4 = Button
button5 = Button
```

Input devices may have up to five buttons.

```
eqButton = /[Button Button /Bool]
```

(eqButton b1 b2) tests if the buttons b1 and b2 are equal.

```
buttonString = /[Button /String]
```

(buttonString b) converts the button b to a string.

6.1.15 Events

```
allocEvent = /[/XEvent]
```

Allocates a new event (specific event values are written into an event by `nextEvent`).

```
nextEvent = /[Display XEvent Sig]
```

`(nextEvent d e)` reads the next event from the display `d` and writes it into `e`, blocking the *whole* Pict system if no event is available.

```
pending = /[Display /Int]
```

`(pending d)` returns the number of events currently available in the input buffer of `d` (if there are no events in the input buffer, `pending` flushes the output buffer and rechecks the status of the input buffer in case some events have become available).

```
isExpose = /[XEvent /Bool]
exposeEventWindow = /[XEvent /Window]
exposeEventX = /[XEvent /Int]
exposeEventY = /[XEvent /Int]
exposeEventW = /[XEvent /Int]
exposeEventH = /[XEvent /Int]
```

If `(isExpose e)` is `true` then the above functions extract the appropriate components of the event `e`. Anything can happen if you use these functions on another kind of event.

```
isButtonPress = /[XEvent /Bool]
isButtonRelease = /[XEvent /Bool]
buttonEventWindow = /[XEvent /Window]
buttonEventX = /[XEvent /Int]
buttonEventY = /[XEvent /Int]
buttonEventButton = /[XEvent /Button]
```

If `(isButtonPress e)` or `(isButtonRelease e)` is `true` then the above functions extract the appropriate components of the event `e`. Anything can happen if you use these functions on another kind of event.

```
isButtonMotion = /[XEvent /Bool]
buttonMotionEventWindow = /[XEvent /Window]
buttonMotionEventX = /[XEvent /Int]
buttonMotionEventY = /[XEvent /Int]
buttonMotionEventTime = /[XEvent /Int]
```

If `(isButtonMotion e)` is `true` then the above functions extract the appropriate components of the event `e`. Anything can happen if you use these functions on another kind of event.

```
isConfigureNotify = /[XEvent /Bool]
configureEventWindow = /[XEvent /Window]
configureEventX = /[XEvent /Int]
configureEventY = /[XEvent /Int]
configureEventW = /[XEvent /Int]
configureEventH = /[XEvent /Int]
```

If `(isConfigureNotify e)` is `true` then the above functions extract the appropriate components of the event `e`. Anything can happen if you use these functions on another kind of event.

```
isKeyPress = /[XEvent /Bool]
isKeyRelease = /[XEvent /Bool]
keyEventWindow = /[XEvent /Window]
keyEventX = /[XEvent /Int]
keyEventY = /[XEvent /Int]
keyEventString = /[XEvent /String]
keyEventKeySym = /[XEvent /KeySym]
```

If `(isKeyPress e)` or `(isKeyRelease e)` is `true` then the above functions extract the appropriate components of the event `e`. Anything can happen if you use these functions on another kind of event.

6.1.16 Event Masks

```
noEventMask = EventMask
keyPressMask = EventMask
keyReleaseMask = EventMask
buttonPressMask = EventMask
buttonReleaseMask = EventMask
enterWindowMask = EventMask
leaveWindowMask = EventMask
pointerMotionMask = EventMask
pointerMotionHintMask = EventMask
button1MotionMask = EventMask
button2MotionMask = EventMask
button3MotionMask = EventMask
button4MotionMask = EventMask
button5MotionMask = EventMask
buttonMotionMask = EventMask
keymapStateMask = EventMask
exposureMask = EventMask
visibilityChangeMask = EventMask
structureNotifyMask = EventMask
resizeRedirectMask = EventMask
substructureNotifyMask = EventMask
substructureRedirectMask = EventMask
focusChangeMask = EventMask
propertyChangeMask = EventMask
colormapChangeMask = EventMask
ownerGrabButtonMask = EventMask
```

Including one of the above mask values in a window's mask indicates an interest in the like-named event.

```
selectInput = /[Display Window EventMask Sig]
```

`(selectInput d win m)` requests that the X server ignore all events in the window `win` except those specified in the event mask `m`. Initially, X will not report any events for a window. Events are reported relative to a window. If a window is not interested in a device event, it usually propagates to the closest ancestor that is interested.

```
mergeMasks = /[EventMask EventMask /EventMask]
```

`(mergeMasks m1 m2)` merge the event masks `m1` and `m2`.

6.1.17 Miscellaneous

```
bell = /[Display Int Sig]
```

`(bell d volume)` rings the bell on the keyboard of the display `d`, if possible. The specified volume is relative to the base volume for the keyboard. The `volume` argument must be in the range `-100` to `100` inclusive. The volume at which the bell rings when the `volume` argument is positive is: $\texttt{base} - ((\texttt{base} * \texttt{volume})/100) + \texttt{volume}$. The volume at which the bell rings when the `volume` argument is negative is: $\texttt{base} + ((\texttt{base} * \texttt{volume})/100)$.

6.2 `X11/Xrm`: X Resource Manager Intrinsics

This library provides an interface to the X resource manager library[2].

[2]This documentation is derived from the X Consortium `Xrm` manual pages

6.2.1 ResourceLine Syntax

The syntax of a resource file is a sequence of resource lines terminated by newline characters or the end of the file. The syntax of an individual resource line is:

ResourceLine	=	Comment | IncludeFile | ResourceSpec | *empty line*
Comment	=	`!` *any character except null or newline*
IncludeFile	=	`#` WhiteSpace `include` WhiteSpace FileName WhiteSpace
FileName	=	*valid filename for operating system*
ResourceSpec	=	WhiteSpace ResourceName WhiteSpace `:` WhiteSpace Value
ResourceName	=	[Binding] {Component Binding} ComponentName
Binding	=	`.` | `*`
WhiteSpace	=	{*space* | *horizontal tab*}
Component	=	`?` | ComponentName
ComponentName	=	NameChar {NameChar}
NameChar	=	`a-z` | `A-Z` | `0-9` | `_` | `--`
Value	=	{*any character except null or unescaped newline*}

Elements separated by vertical bar (|) are alternatives. Curly braces ({...}) indicate zero or more repetitions of the enclosed elements. Square brackets ([...]) indicate that the enclosed element is optional. `Teletype` font is used for literal characters.

IncludeFile lines are interpreted by replacing the line with the contents of the specified file. The word `include` must be in lowercase. The file name is interpreted relative to the directory of the file in which the line occurs (for example, if the file name contains no directory or contains a relative directory specification).

If a ResourceName contains a contiguous sequence of two or more Binding characters, the sequence will be replaced with single `.` character if the sequence contains only `.` characters; otherwise, the sequence will be replaced with a single `*` character.

A resource database never contains more than one entry for a given ResourceName. If a resource file contains multi- ple lines with the same ResourceName, the last line in the file is used.

Any white space characters before or after the name or colon in a ResourceSpec are ignored. To allow a Value to begin with white space, the two-character sequence `\space` (backslash followed by space) is recognized and replaced by a space character, and the two-character sequence `\tab` (backslash followed by horizontal tab) is recognized and replaced by a horizontal tab character. To allow a Value to contain embedded newline characters, the two-character sequence `\n` is recognized and replaced by a newline character. To allow a Value to be broken across multiple lines in a text file, the two- character sequence `\newline` (backslash followed by newline) is recognized and removed from the value. To allow a Value to contain arbitrary character codes, the four-character sequence `\nnn`, where each `n` is a digit character in the range of `0-7`, is recognized and replaced with a single byte that contains the octal value specified by the sequence. Finally, the two-character sequence `\\` is recognized and replaced with a single backslash.

6.2.2 Resource Matching

The algorithm for determining which resource database entry matches a given query is the heart of the resource manager. All queries must fully specify the name and class of the desired resource (use of the characters `*` and `?` are not permitted). The library supports up to 100 components in a full name or class. Resources are stored in the database with only partially specified names and classes, using pattern matching constructs. An asterisk (`*`) is a loose binding and is used to represent any number of intervening components, including none. A period (`.`) is a tight binding and is used to separate immediately adjacent components. A question mark (`?`) is used to match any single component name or class. A database entry cannot end in a loose binding; the final component (which cannot be the character `?`) must be specified. The lookup algorithm searches the database for the entry that most closely matches (is most specific for) the full name and class being queried. When more than one database entry matches the full name and class, precedence rules are used to select just one.

The full name and class are scanned from left to right (from highest level in the hierarchy to lowest), one component at a time. At each level, the corresponding component and/or binding of each matching entry is determined, and these matching components and bindings are compared according to precedence rules. Each of the rules is applied at each level before moving to the next level, until a rule selects a single entry over all others. The rules, in order of precedence, are:

1. An entry that contains a matching component (whether name, class, or the character `?`) takes precedence over entries that elide the level (that is, entries that match the level in a loose binding).

2. An entry with a matching name takes precedence over both entries with a matching class and entries that match using the character `?`. An entry with a matching class takes precedence over entries that match using the character `?`.

3. An entry preceded by a tight binding takes precedence over entries preceded by a loose binding.

6.2.3 Types

```
#XrmDatabase
```

The type of a resource database.

6.2.4 Resource Database Operations

```
initialize = /[Sig]
```

Initializes the resource manager. This function must be called before any other `xrm` functions are used.

```
string = /[Display Exn /String]
```

`(string d h)` gets the `RESOURCE_MANAGER` property from the display `d`'s root window of screen zero. Calls the error handler `h` if no such property exists.

```
getDatabase = /[Display Exn /XrmDatabase]
```

`(getDatabase d h)` returns the database associated with the display `d`. Calls the handler `h` if a database has not yet been set.

```
fromString = /[String /XrmDatabase]
```

`(fromString s)` converts `s` into a new database. The string `s` should contain a sequence of entries in valid ResourceLine format (as described above).

```
destroy = /[XrmDatabase Sig]
```

`(destroy db)` deallocates the database `db`.

```
combine = /[XrmDatabase XrmDatabase Bool /XrmDatabase]
```

`(combine src dest o)` merges `src` into `dest`. If the same specifier is used for an entry in both databases, the entry in the `src` will replace the entry in the `dest` if `o` is `true`; otherwise, the entry in `src` is discarded. `src` is destroyed by the merge.

```
getResource = /[XrmDatabase String String /(Opt String)]
```

`(getResource db name class)` gets the specified resource, returning `(opt.yes s)` if the string `s` was found, and `opt.no` otherwise.

Chapter 7

Composable Pictures

7.1 Picture/Basic: Basic Picture Primitives

All the operations in this module are re-exported from the module Picture/Picture (where they are documented).

```
null = Picture
point = /[Int Int /Picture]
line = /[Int Int Int Int /Picture]
hLine = Picture
vLine = Picture
quad = /[Int Int Int Int Int Int Int Int /Picture]
fillQuad = /[Int Int Int Int Int Int Int Int /Picture]
rect = /[Int Int Int Int /Picture]
fillRect = /[Int Int Int Int /Picture]
clickable = /[Int Int Int Int /[Button Bool Int Int Int Int] /Picture]
spacer = /[Int Int Int Int /Picture]
ellipse = /[Int Int Int Int /Picture]
fillEllipse = /[Int Int Int Int /Picture]
text = /[String /Picture]
pixmap = /[String Exn /Picture]
```

7.2 Picture/Combine: Combining Pictures

All the operations in this module are re-exported from the module Picture/Picture (where they are documented).

```
add = /[Picture Picture /Picture]
beside = /[Picture Picture /Picture]
above = /[Picture Picture /Picture]
scaledAbove = /[Int Int Picture Picture /Picture]
scaledBeside = /[Int Int Picture Picture /Picture]
background = /[Color Picture /Picture]
border = /[Int Int Picture /Picture]
```

7.3 Picture/Other: Other Pictures

All the operations in this module are re-exported from the module Picture/Picture (where they are documented).

```
#Slider =
  [w=Int h=Int min=Int max=Int range=Int pos=Int posChanged=/[Int Sig]]
```

```
ref = /[Picture ?[Picture Sig] /Picture]
stringRef = /[String ?[String Sig] /Picture]
intRef = /[Int ?[Int Sig] /Picture]
array = /[Int Int (Array Picture) ?[Int Int (Array Picture) Sig] /Picture]
array2 = /[Int Int (Array2 Picture) ?[Int Int (Array2 Picture) Sig] /Picture]
button = /[String /[Sig] /Picture]
quitButton = Picture
slider = /[Slider /[Picture (Ref Slider)]]
```

7.4 `Picture/Package`: Unpackaging Pictures

This module provides various operations which unpackage an active picture and call one of its methods. It should not be necessary to use these operations in user programs, unless you are constructing your own pictures from scratch.

```
rotate = /[Active Rot /Active]
rot90  = /[Active /Active]
rot180 = /[Active /Active]
rot270 = /[Active /Active]
```

`(rotate ac a)` rotates the active picture `ac` by `a` degrees. We provide `rot90`, `rot180` and `rot270` as special cases of `rotate`, since it is often possible to implement rotations by these angles more efficiently.

```
scale = /[Active Int Int Int Int /Active]
```

`(scale ac sx dx sy dy)` scales a active picture `ac` by `dx/sx` in the x-axis, and `dy/sy` in the y-axis.

```
move = /[Active Int Int /Active]
```

`(move ac dx dy)` moves the active picture `ac` by `dx` along the x-axis and `dy` along the y-axis.

```
button = /[Active CX Int Int Button Bool Int /Bool]
```

`(button ac cx x y b ud t)` sends a button press/release message to the active picture `ac`. The coordinates of the button press/release are `[x y]`, the button pressed is `b`, `ud` indicates whether the button was released or pressed. The integer `t` indicates (in pixels) how close a button press must be to an object before it is considered a hit.

```
boundary = /[Active /[Int Int Int Int]]
```

`(boundary ac)` computes the bounding box of the active picture `ac`.

```
render = /[Active CX Sig]
```

`(render ac cx)` renders the active picture `ac` using the context `cx`.

```
init = /[Picture CX /Active]
```

`(init p cx)` initialises the picture `p` using the context `cx`, return a new active picture.

```
constant = /[Picture /Bool]
```

`(constant p)` returns `true` if the picture `p` has no updatable components. This attribute is used to avoid needlessly reconfiguring components of pictures which have not changed.

```
reconfigure = /[Active CX /Active]
```

`(reconfigure ac cx)` reconfigures the active picture `ac`. This operation is used when a picture has signalled that it wishes to change its bounding box, or rearrange some of its sub-pictures.

```
delete = /[Active Sig]
```

`(delete ac)` frees any resources consumed by the active picture `ac`.

7.5 `Picture/Picture`: Picture Operations

This module does not define any new operations in itself, it simply re-exports all the user-level operations defined in the picture libraries. If you wish to write new picture primitives (rather than composing new pictures from the existing primitives), you will probably need to use picture operations which are not re-exported here (cf. Sections 7.9, 7.4 and 7.8).

(The `Picture` and `Server` types referred to in the following type definitions are defined in Section 7.8.)

7.5.1 Server Connection

```
server = /[String String /Server]
```

`(server d a)` opens a connection to the display `d` (an empty string means connect to the default display). The string `a` is the application name (used for resource database lookup).

7.5.2 Picture Primitives

```
null = Picture
```

The null picture.

```
point = /[Int Int /Picture]
```

`(point x y)` creates a point picture (a single pixel). Scaling a point will affect its coordinates, but not its size.

```
line = /[Int Int Int Int /Picture]
```

`(line x0 y0 x1 y1)` creates a line from `[x0 y0]` to `[x1 y1]`. Scaling a line will affect its endpoints but not its thickness.

```
hLine = Picture
vLine = Picture
```

Unit length horizontal and vertical lines (equivalent to `(line 0 0 1 0)` and `(line 0 0 0 1)` respectively).

```
quad = /[Int Int Int Int Int Int Int Int /Picture]
fillQuad = /[Int Int Int Int Int Int Int Int /Picture]
```

`(quad x0 y0 x1 y1 x2 y2 x3 y3)` creates a closed polygon with four vertices. Scaling such a polygon will affect its shape but not the thickness of its lines. `fillQuad` creates a filled polygon.

```
rect = /[Int Int Int Int /Picture]
fillRect = /[Int Int Int Int /Picture]
```

`(rect x y w h)` creates a rectangle whose leftmost corner is at `[x y]`, and whose width and height are `w` and `h` respectively. `fillRect` creates a filled rectangle.

```
clickable = /[Int Int Int Int /[Button Bool Int Int Int Int] /Picture]
```

```
spacer = /[Int Int Int Int /Picture]
```

`(spacer x y w h)` creates a space whose leftmost corner is at `[x y]`, and whose width and height are `w` and `h` respectively.

```
ellipse = /[Int Int Int Int /Picture]
fillEllipse = /[Int Int Int Int /Picture]
```

`(ellipse x y w h)` creates an ellipse of width `w` and height `h`, centred at `[x y]`. `fillEllipse` creates a filled ellipse.

```
text = /[String /Picture]
```

`(text s)` creates a text picture using the string `s`. Note that we do not currently implement font rotation or scaling, so scaling or rotating a text picture will not affect the size or orientation of the text, just the drawing origin. The string `s` should not contain newlines.

```
pixmap = /[String Exn /Picture]
```

`(pixmap f h)` loads a pixmap in ppm format from the file `f`, calling the error handler `h` if the `f` could not be opened or `f` is not a valid ppm file. Note that we do not currently implement pixmap rotation or scaling, so scaling or rotating a pixmap will not affect the size or orientation of the pixmap, just the drawing origin.

7.5.3 Picture Transformations

```
rotate = /[Picture Int /Picture]
```

`(rotate p x)` rotates the picture `p` by `x` degrees.

```
scale = /[Picture Int Int Int Int /Picture]
```

`(scale p sx dx sy dy)` scales the picture `p` by `dx/sx` in the x-axis and `dy/sy` in the y-axis.

```
scaleTo = /[Picture Int Int /Picture]
```

`(scaleTo p w h)` scales the picture `p` so that it has width `w` and height `h`.

```
move = /[Picture Int Int /Picture]
```

`(move p x y)` moves the picture `p` along the x-axis by `x` pixels and along the y-axis by `y` pixels.

```
fg = /[Color Picture /Picture]
font = /[Font Picture /Picture]
```

Functions to change the default foreground color and font of a picture.

```
center    = /[Picture /Picture]
north     = /[Picture /Picture]
northeast = /[Picture /Picture]
east      = /[Picture /Picture]
southeast = /[Picture /Picture]
south     = /[Picture /Picture]
southwest = /[Picture /Picture]
west      = /[Picture /Picture]
northwest = /[Picture /Picture]
```

Moves the origin of the given picture to the appropriate point on the picture's bounding box.

```
catchHit  = /[Picture /[CX Button Bool Sig] /Picture]
```

`(catchHit p a)` creates a picture with the same visual appearance as `p`. The modified picture catches any button presses which hit `p`, and sends a message to `a`, notifying it of the button that was pressed (the boolean value indicates whether the button was released or pressed).

```
quitOnHit = /[Picture /Picture]
```

`(quitOnHit p)` creates a picture with the same visual appearance as `p`. The modified picture catches any button presses which hit `p`, and responds by sending a quit request to the picture's parent.

7.5.4 Combining Pictures

```
add = /[Picture Picture /Picture]
```

`(add p q)` overlays the pictures `p` and `q`. The bounding box for the combined picture is the least bounding box which includes the bounding boxes of `p` and `q`. This function can be combined with the transposition functions described above to provide a number of other useful picture combinators.

```
beside = /[Picture Picture /Picture]
```

`(beside p q)` is an example of a useful combination of the above transposition functions and the picture overlay function `add`. It transforms `p` and `q` so that `p`'s origin is at the top right of its bounding box, and `q`'s origin is at the top left of its bounding box. Thus, since `add` aligns the origins of `p` and `q`, this has the effect of juxtaposing `p` and `q` and aligning the tops of their bounding boxes.

```
above = /[Picture Picture /Picture]
```

`(above p q)` juxtaposes `p` and `q` vertically (aligning the left-hand sides of their bounding boxes).

```
scaledAbove = /[Int Int Picture Picture /Picture]
```

`(scaledAbove n1 n2 p1 p2)` composes `p1` and `p2` vertically, scaling `p1` so that it is `n1/n2` times the height of `p2` and either scaling up `p1` so that it is the same width as `p2`, or vice versa.

```
scaledBeside = /[Int Int Picture Picture /Picture]
```

`(scaledAbove n1 n2 p1 p2)` composes `p1` and `p2` horizontally, scaling `p1` so that it is `n1/n2` times the width of `p2` and either scaling up `p1` so that it is the same height as `p2`, or vice versa.

```
background = /[Color Picture /Picture]
```

`(background c p)` puts `p` on top of a rectangle of color `c` and with the same size as `p`.

```
border = /[Int Int Picture /Picture]
```

`(border w m)` Adds a rectangular border of width `w` around the picture `p`, leaving a margin of `m` pixels around `p`'s boundary.

7.5.5 Other Pictures

```
ref = /[Picture ?[Picture Sig] /Picture]
```

`(ref p u)` creates an updatable picture whose initial appearance is `p`, and whose appearance can be modified using the channel `u`. For example, `(u q)` updates `(ref p u)` with the new picture `q`.

```
stringRef = /[String ?[String Sig] /Picture]
intRef = /[Int ?[Int Sig] /Picture]
```

`(stringRef s u)` creates an updatable picture which displays the string `s`. The channel `u` can be used to update the string which is being displayed. `intRef` is similar to `stringRef` except that it displays an integer value.

```
array = /[Int Int (Array Picture) ?[Int Int (Array Picture) Sig] /Picture]
array2 = /[Int Int (Array2 Picture) ?[Int Int (Array2 Picture) Sig] /Picture]
```

`(array sz gap a u)` create an updatable picture which displays a horizontal array of pictures `a`. Each picture in `a` is displayed in an area of size `sz`, with `gap` pixels between each element. `array2` is similar to `array`, except that it creates a two-dimensional grid of pictures.

```
button = /[String /[Sig] /Picture]
```

`(button label action)` creates a simple button picture using the given a `label` string. The `action` channel is signalled on whenever a mouse button is clicked on the picture.

```
quitButton = Picture
```

A simple quit button which, whenever a mouse button is clicked on the picture, sends a quit request to its parent picture. If none of the button's parents handle this quit request, this has the effect of quitting the top-level window containing the quit button.

```
slider = /[Slider /[Picture (Ref Slider)]]
```

`(slider s)` returns a slider which can represent a range of values between `s.min` and `s.max` (inclusive). The initial range of values is `s.pos...(s.pos+s.range)` (inclusive). The `(Ref Slider)` can be used to modify the aforementioned values. The `s.posChanged` channel is signalled on whenever the user changes the position of the slider.

7.6 `Picture/Server`: Server Connection

All the operations in this module are re-exported from the module `Picture/Picture` (where they are documented).

```
def server (displayName:String appName:String) : Server
```

7.7 `Picture/Transform`: Picture Transformations

All the operations in this module are re-exported from the module `Picture/Picture` (where they are documented).

```
rotate = /[Picture Int /Picture]
scale = /[Picture Int Int Int Int /Picture]
scaleTo = /[Picture Int Int /Picture]
move = /[Picture Int Int /Picture]
fg = /[Color Picture /Picture]
font = /[Font Picture /Picture]
center    = /[Picture /Picture]
north     = /[Picture /Picture]
northeast = /[Picture /Picture]
east      = /[Picture /Picture]
southeast = /[Picture /Picture]
south     = /[Picture /Picture]
southwest = /[Picture /Picture]
west      = /[Picture /Picture]
northwest = /[Picture /Picture]
catchHit  = /[Picture /[CX Button Bool Sig] /Picture]
quitOnHit = /[Picture /Picture]
```

7.8 `Picture/Types`: Picture Interface Types

The `Picture` type is unimportant unless you wish to create your own picture primitives.

7.8.1 Resource Manager

```
type ResourceManager = [
```

```
newFont = /[String Exn /Font]
```

(newFont n h) allocates the font named n, calling the given error handler h if the font could not be found. newFont caches the results of font lookups (to avoid the long delays caused by repeatedly querying the X server).

```
newColor = /[String Exn /Color]
newColorRGB = /[Int Int Int (Handler [Int Int Int]) /Color]
```

(newColor n h) allocates the color named n, calling the given error handler h if the color could not be found. newColor caches the results of color lookups. (newColorRGB r g b h) behaves similarly, except that the required color is specified as a triple of RGB intensities (in the range 0 to 65535).

```
resource = /[String Exn /String]
```

(resource n h) looks up the resource name n in the resource database, calling the given error handler if no matching resource could be found. It first searches the RESOURCE_MANAGER string, and if it cannot find a matching entry it searches the application-specific default resources (which are specified we open a connection to the server using initResourceManager).

```
colorResource = /[String String /Color]
```

Looks up the resource name res in the resource database. If a matching resource is found, we attempt to load that color. If no matching resource is found or if the color specified in the resource cannot be loaded, we load the color default and return that instead (in this case, if default is not a valid color name we generate a runtime error).

```
fontResource = /[String String /Font]
```

Looks up the resource name res in the resource database. If a matching resource is found, we attempt to load that font. If no matching resource is found or if the font specified in the resource cannot be loaded, we load the font default and return that instead (in this case, if default is not a valid font name we generate a runtime error).

```
intResource = /[String Int /Int]
```

Looks up the resource name res in the resource database. If a matching resource is found, we attempt to convert the resource string to an integer. If no matching resource is found or the string specified in the resource is not an integer, we return default value x.

```
boolResource = /[String Bool /Bool]
```

Looks up the resource name res in the resource database. If a matching resource is found, we attempt to convert the resource string to a boolean. If no matching resource is found or the string specified in the resource is not a boolean, we return default value b.

```
]
```

7.8.2 Contexts

```
type CX =
  (ResourceManager with
   display = Display
   screen  = ScreenNum
   root    = Window
   window  = Window
   font = Font
   fg = Color
   bg = Color
   gc = GC
   useGc   = /[/[GC Sig] Sig]
   quit    = ![]
   changed = ![])
```

7.8.3 Boundaries

```
type Bounds = [Int Int Int Int]
```

7.8.4 Methods

```
type (M I R) =
  [constant = Bool
   reconfigure = /[I R CX /R]
   delete = /[R Sig]
   boundary = /[R /Bounds] {- Combine with init method -}
   move = /[R Int Int /R]
   scale = /[R Int Int Int Int /R]
   rotate = /[R Rot /R]
   rot90 = /[R /R]
   rot180 = /[R /R]
   rot270 = /[R /R]
   button = /[R CX Int Int Button Bool Int /Bool]
   render = /[R CX Sig]
   init = /[I CX /R]]
```

7.8.5 Pictures

```
type Picture = [#I #R (M I R) I]
```

Values of type `Picture` may be scaled, moved, rotated, or rendered (onto some X window or pixmap). We also provide various operators for building composite pictures.

```
type Active = [#I #R (M I R) I R]
```

7.8.6 Servers

```
type Controller =
  [map   = /[Sig]
   unmap = /[Sig]
   quit  = /[Sig]]

type Server =
  (ResourceManager with
   open = /[String Picture Sig]
   openControlled = /[String Picture /Controller])
```

7.9 `Picture/Utils`: Picture Utilities

```
val [#Rot utils:[
make = /[Int /Rot]
rotx = /[Int Int Rot /Int]
roty = /[Int Int Rot /Int]
]]
```

Bibliography

[Nye90] Adrian Nye. *Xlib Programming Manual*, volume 1. O'Reilly & Associates, Inc., 1990.

[SG92] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.