Nomadic Pict Language Libraries
Release 1.0-alpha

Paweł T. Wojciechowski

University of Cambridge
Computer Laboratory
New Museums Site
Cambridge CB2 3QG
England
`Pawel.Wojciechowski@cl.cam.ac.uk`

December 18, 2000

Contents

Copying

Foreword

This document describes Nomadic Pict libraries, which can be used in programs executed by the Nomadic Pict runtime system. Our system supports a large subset of Pict standard libraries. We describe these libraries in Chapter 3. The chapter also contains functions which are characteristic for Nomadic Pict, such as runtime system utilities, described in Section 3.16. In Chapter 5, we describe a simple graphical library (derived from Objective Caml). In this document we use extracts from the original Pict documentation [PT97], by courtesy of Benjamin C. Pierce.

Chapter 1

Library Conventions

The Nomadic Pict language uses the same library conventions as Pict. The 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, $\mathtt{X}_1, \ldots, \mathtt{X}_n$, which may either be abstract or opaque types, and a single top-level value identifier `module`, which is bound to some record of operations $\mathtt{op}_1, \ldots, \mathtt{op}_n$. The value identifier is usually the lowercase version of the library filename (for example, `Nstd/List` introduces the value identifier `list`).

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 `Nstd/List`, the type `List` is directly available to the programmer and the value identifier `list` is bound to a record of operations on lists. This allows us to use qualified names for our list operations. For instance, `list.nil` is a function which returns the empty list.

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 hash 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

`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

`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, `list.make` builds a fresh list (given the size of the list 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, `(list.unsafeCar l)` returns the head of the list `l`, just like `(list.car l)`, but it avoids doing any checking if `l` is the empty list. 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 currently only two groups of library modules (although more libraries can be supported easily if required):

`Nstd` Standard library modules which we expect them to exist on any platform which supports Nomadic Pict.

`Graphics` High-level interfaces to `X11` facilities using the graphics library of Objective Caml.

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

Chapter 3

Standard Libraries

3.1 Nstd/Bool: Booleans

3.1.1 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.

3.1.2 Operations

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

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

3.1.3 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.

3.1.4 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.

3.1.5 Printing

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

(pr b) prints b on stdout.

3.1.6 Abbreviations

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

3.2 Nstd/Char: Characters

3.2.1 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.

3.2.2 Conversion

```
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.

3.2.3 Printing

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

(pr c) prints the character c on stdout.

3.3 Nstd/Cmp: Comparison Results

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.3.4 Conversion

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

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

3.3.5 Printing

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

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

3.4 `Nstd/Error`: Error Handling

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

```
exit = /[#X Int /X]
```

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

```
fatal = /[#X String /X]
```

`(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.

3.5 `Nstd/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."

3.5.1 Types

```
#Handler = \X = /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.

3.5.2 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 #Y (Handler X) X /Y]
```

`(raise h x)` sends the value `x` to the handler `h` and blocks the caller. (The result type `Y` is polymorphic so that `raise` can be used in a context where any type of result is expected.)

```
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.

3.6 Nstd/Int: Integers

3.6.1 Types

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

3.6.2 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`.

3.6.3 Bitwise Operations

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

Bitwise and, or, exclusive-or and negation.

3.6.4 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.

3.6.5 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.

3.6.6 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.

3.6.7 Printing

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

(pr x) prints x on stdout.

3.6.8 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
```

3.7 Nstd/List: Lists

3.7.1 Types

```
#List < Top2
```

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

3.7.2 Creation

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

(nil) returns 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).

3.7.3 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).

3.7.4 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`.

3.7.5 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`.

3.7.6 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.

3.7.7 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 3.14).

3.7.8 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 3.3).

3.7.9 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`.

3.7.10 Abbreviations

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

3.8 `Nstd/Map`: Maps

This module implements applicative association tables (i.e. finite maps or dictionaries), given a total ordering function over the keys. All operations over maps are purely applicative (no side-effects). The implementation uses lists.

3.8.1 Types

```
#Map < Top3
```

`(Map K V)` is the type of maps from type `K` to type `V`.

3.8.2 Creation and removal

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

`(make m eq)` returns the empty map. `eq` is a comparing function over the keys; `(eq e1 e2)` returns `true` if `e1` and `e2` are equal and `false` otherwise.

```
add = /[#K #V (Map K V) K V /(Map K V)]
```

`(add m k v)` returns a map containing the same binding as `m`, plus a binding of `k` to `v`. If `k` was already bound in `m`, its previous binding disappears.

```
remove = /[#K #V (Map K V) K /(Map K V)]
```

`(remove m k)` returns a map containing the same binding as `m`, except for `k` which is unbound in the returned map.

3.8.3 Interrogation

```
present = /[#K #V (Map K V) K /Bool]
```

`(present m k)` returns `true` if there is binding of `k` in `m`, or `false` otherwise.

```
lookup = /[#K #V (Map K V) K /[Found>V  NotFound>[]]]
```

`(lookup m k)` returns a variant of current binding of `k` in `m`, or a null variant `[NotFound>[]]` if no such binding exists.

3.8.4 Iteration

```
iter = /[#K #V (Map K V) /[K V /[]] /[]]
```

`(iter m f)` applies `f` to all bindings in map `m`, discarding the results. `f` receives the key as first argument, and the associated values as second argument. The order in which the bindings are passed to `f` is unspecified.

```
fold = /[#K #V #W (Map K V) /[K V W /W] W /W]
```

`(fold m f a)` computes `(f kn vn ... (f k1 v1 a) ... )`, where `k1` ... `kn` are the keys of all bindings in `m`, and `v1` ... `vn` are the associated data. The order in which the bindings are presented to `f` is not specified.

3.9 `Nstd/Misc`: Miscellaneous Utilities

3.9.1 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.

```
#Top2 = \X = Top
#Top3 = \X Y = Top
```

`Top2` and `Top3` are convenient abbreviations for the type `Top` at kinds `(Type->Type)` and `(Type Type->Type)`.

3.9.2 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`.

3.9.3 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`.

3.9.4 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 `[]`.

3.9.5 Function Composition

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

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

3.9.6 Printing

The following printing functions just use the standard buffered input/output operations provided by Objective Caml. 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 [*s:Io?*]). ?

```
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.)

3.9.7 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 flush=misc.flush
val print = misc.print
val printi = misc.printi
```

3.10 `Std/Opt`: Optional Values

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

3.10.1 Types

```
#Opt < Top2
```

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

3.10.2 Construction

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

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

3.10.3 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).

3.10.4 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`.

3.11 `Nstd/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 `Nstd/Int` and `Nstd/String`.

3.11.1 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.

3.12 `Std/Ref`: Reference Cells

3.12.1 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.

3.12.2 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.

3.13 `Nstd/Set`: Sets

This module implements sets of data. All operations over sets are purely applicative (no side-effects). The implementation uses lists.

3.13.1 Types

```
#Set < Top2
```

`(Set K)` is the type of sets of type `K`.

3.13.2 Creation and removal

```
make = /[#K /[K K /Bool] /(Set K)]
```

`(make s eq)` returns the empty set. `eq` is a comparing function over the keys; `(eq e1 e2)` returns `true` if `e1` and `e2` are equal and `false` otherwise.

```
add = /[#K (Set K) K /(Set K)]
```

`(add s k)` returns a set containing the same data as `m`, plus a value `k`. If `k` was already in `m`, the operation does not have any effect.

```
remove = /[#K (Set K) K /(Set K)]
```

`(remove m k)` returns a set containing the same data as `m`, except for `k` which is removed from the returned set.

3.13.3 Interrogation

```
present = /[#K (Set K) K /Bool]
```

`(present m k)` returns `true` if there is `k` in `m`, or `false` otherwise.

3.13.4 Iteration

```
iter = /[#K (Set K) /[K /[]] /[]]
```

(`iter m f`) applies `f` to all values in set `m`, discarding the results. The order in which the values are passed to `f` is unspecified.

```
fold = /[#K #W (Set K) /[K W /W] W /W]
```

(`fold m f a`) computes (`f kn ... (f k1 a) ...`), where `k1` ... `kn` are all values in `m`. The order in which the values are presented to `f` is not specified.

3.14 `Nstd/Stream`: Streams

3.14.1 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.

3.14.2 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`.

3.14.3 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`.

3.15 `Nstd/String`: Strings

3.15.1 Types

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

3.15.2 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`.

3.15.3 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` $\geq$ `0`, `start` $\geq$ `0` and `start` $+$ `len` $\leq$ `(size s)`.

3.15.4 Comparison

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

Returns a single value indicating the ordering of `s` and `t` (cf. Section 3.3).

```
==$ = /[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`.

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

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

3.15.5 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"`.

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

`(toInt s h)` converts `s` to an integer. Calls the error handler `h` if if `s` is not a valid representation of an integer.

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

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

3.15.6 Printing

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

(`pr s`) prints the string `s` on stdout.

3.15.7 Abbreviations

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

3.16 `Nstd/Sys`: Runtime System Utilities

3.16.1 Types

The types `Site` and `Agent` are built in and cannot be redefined.

3.16.2 Comparison

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

(`== a b`) tests for structural equality of `a` and `b`. Mutable structures are equal if and only if their current contents are structurally equal, even if the two mutable objects are not the same physical object. Equality between cyclic data structures may not terminate. Note that all strings in Nomadic Pict have unique internal representations, so for comparison of strings one should use `==$`, not `sys.==`.

```
<> = /[#X X X /Bool]
```

Negation of `==`

```
eqa = /[Agent Agent /Bool]
```

(`eqa a b`) is an abbreviation for (`sys.== #Agent a b`)

```
eqs = /[Site Site /Bool]
```

(`eqs a b`) is an abbreviation for (`sys.== #Site a b`)

3.16.3 Channel Operations

```
chan_name = /[#X ^X /String]
```

(`chan_name c`) returns a globally unique ID of channel `c` converted to String.

3.16.4 Site Operations

```
addr_of_site = /[Site /String]
```

(`addr_of_site s`) returns Internet address of site `s` converted to String.

```
port_of_site = /[Site /Int]
```

(`port_of_site s`) returns a port number of site `s`.

```
site_name = /[Site /String]
```

(`site_name s`) returns an official name of site `s` (adddress+port).

3.16.5 Agent Operations

```
this_agent = /[/Agent]
```

(this_agent) returns the name of type Agent of the agent which invoked the function.

```
agent_id = /[/String]
```

(agent_id) returns agent ID converted to String.

```
is_here = /[Agent /Bool]
```

(is_here a) returns true if agent a is on this site or false otherwise.

3.16.6 Config Operations

```
gethostname = /[/String]
```

gethostname returns the local host name (e.g. "glia.cl.cam.ac.uk").

```
this_site = /[/Site]
```

(this_site) returns the local site name of type Site.

```
get_site = /[Int /Site]
```

(get_site i) reads the i-th pair (String,Int) of IP address and port number from the config file, and returns value of type Site. Rise failure if i is outside the range from 0 to a total number of sites -1.

```
get_next_site = /[/Site]
```

(get_next_site) reads the first non-read pair (String,Int) of IP address and port number from the config file, and returns value of type Site. Start reading the file from the beginning if all pairs have been read by previous invocations of (get_next_site). Rise failure if the format of config file is not recognised. Warning: make sure that values returned by the function are actually used in your program, otherwise the compiler can decide to remove get_next_site which might lead to confusion about which values are returned by following invocations of this function [1].

3.16.7 Runtime System and I/O Operations

```
time = /[/Int]
```

(time) returns the current time since 00:00:00 GMT, Jan. 1, 1970, in seconds.

```
sleep = /[Int Sig]
```

Stop execution of the current parallel process for the given number of seconds

```
saveall = /[String /String]
```

(saveall prefix) saves code and execution state of all agents currently executed on a local site to a fresh file in the temporary directory /tmp or the directory specified by a variable TMPDIR (if set). The file name, guaranteed to be different from any other file names in the directory, is formed by concatenating prefix with some freshly created name and suffix ".svp". The file is created with permissions 0o600, i.e. readable and writable only by the file owner. The function returns an absolute name of the file (i.e. including the path) but without the suffix.

[1] The compiler does static code optimisation and partial evaluation and may arbitrarily decide to remove parts of the program which are not used or can be evaluated statically.

```
saveall_unsafe = /[String Sig]
```

(saveall_unsafe filename) saves code and execution state of all agents currently executed on a local site to a file created by concatenating filename with suffix ".svp". If the file exists then it is is overwritten without warning.

```
load = /[String Sig]
```

(load filename) loads agents from a file created by concatenating filename with suffix ".svp". The runtime system will continue the execution of the agents from a savepoint in which the agents have been stored to the file using saveall or safeall_unsafe. The load is desctructive, i.e. all agents that were present on a site when executing load are destroyed.

```
read_line = /[/String]
```

(read_line) flushes standard output, then reads characters from standard input until a newline character is encountered. Return the string of all characters read, without the newline character at the end.

```
read_int = /[/Int]
```

(read_int) flushes standard output, then reads one line from standard input and convert it to an integer. Raises failure if the line read is not a valid representation of an integer.

```
read = /[String /String]
```

(read s) prints s on standard output, then reads characters from standard input until a newline character is encountered. Return the string of all characters read, without the newline character at the end.

```
readi = /[String /Int]
```

(readi s) prints s on standard output, then reads one line from standard input and convert it to an integer. Raises failure if the line read is not a valid representation of an integer.

3.16.8 Trader Operations

We assume that one Nomadic Pict server has been chosen to be a trader. The address and port number of this server are passed in a command line used to compile and execute programs on other sites.

```
publish = /[String Dyn Sig]
```

(publish key v) contacts the Nomadic Pict trader and publishes a mapping from keyword key of type String to v of type Dyn.

```
subscribe = /[String Agent /Dyn]
```

(subscribe key a) contacts the Nomadic Pict trader and subscribes for value of type Dyn identified by keyword key; the function returns the value mapped by key if it exists or blocks until the value is published. The subscribed value will be sent to agent a.

```
publish_this_site = /[String Sig]
```

(publish_this_site key v) contacts the Nomadic Pict trader and publishes the value of the local site identified by keyword key of type String.

```
subscribe_site = /[String Agent /Site]
```

(subscribe_site key a) contacts the Nomadic Pict trader and subscribes for the value of type Site identified by keyword key; the function returns the value mapped by key if it exists, or blocks until the value is published. The subscribed value will be sent to agent a.

3.16.9 Abbreviations

```
val this_site=sys.this_site
val get_site=sys.get_site
val get_next_site=sys.get_next_site
val publish=sys.publish
val subscribe=sys.subscribe
val publish_this_site=sys.publish_this_site
val subscribe_site=sys.subscribe_site
```

Chapter 4

Distributed Events Library

4.1 `Events/Event`: Distributed events

This library implements distributed events and a composite event language for manipulating composite event expressions (such as a sequence of event types, etc.). A *distributed event* in Nomadic Pict is an asynchronous time-stamped message which can be sent to agents which subscribed for the event type on channels carrying values of abstract type `Event`. See [Woj00] and an example program in the distribution for details.

4.1.1 Types

```
#Event < Top2
```

`(Event X)` is the type of events of elements of type `X`.

4.1.2 Creation

```
address   = /[Agent Site /Address]
```

`(address a s)` returns the address of agent `a` at site `s`.

```
engine    = /[Site Time /Address]
```

`(engine s t)` creates a new event mediator agent at site `s` and returns its address. The mediator will delay any event notifications for `t` seconds so that all events which are generated by distributed event sources at roughly the same time will have chance to be received by the mediator and properly ordered *before* making notifications to the clients that subscribed for these events (this is important in the case of any composite events which depend on a total order). It is assumed that the local clocks at event sources are synchronised and the maximum communication delay between event source and the mediator is less than `t`.

4.1.3 Event source operations

```
declare   = /[#X ^(Event X) Address]
```

`declare![c a]` declares a new event channel `c` at mediator `a`.

```
submit    = /[#X Time ^(Event X) X Address]
```

`submit![t c e a]` submits element `e` of event channel `c` at mediator `a` (with timestamp `t`).

4.1.4 Event subscriber operations

```
register  = /[#X ^(Event X) Address Address]
```

`register![c m a]` subscribes at mediator `m` for events sent on event channel `c`. The events are assumed to be delivered to address `a`.

4.1.5 Discarding events

```
discard   = /[#X ^(Event X) Address]
```

`discard![c m]` causes mediator `m` to discard event notifications on event channel `c` to all clients that ever obtained channel `c` (e.g. by scope extrusion).

4.1.6 Interrogation

```
timestamp = /[#X (Event X) /Time]
```

`(timestamp e)` returns the timestamp of event `e`.

```
number    = /[#X (Event X) /Int]
```

`(number e)` returns the sequence number of event `e` generated by the mediator.

```
value     = /[#X (Event X) /X]
```

`(value e)` returns the element of type `X` of event `e`.

4.1.7 Printing

```
pr = /[#X (Event X) /[Time Int X Sig] Sig]
```

`(print e prx)` prints the event `e` using the supplied printing function `prx`, where the expected arguments of `prx` are: the event timestamp, sequence number, and a value of event `e`.

4.1.8 Composite event language operations

```
inclusiveor  = /[#X #Y ^(Event X) ^(Event Y) Address Address /^(Event [X Y])]
```

`(inclusiveor c1 c2 m a)` returns a new fresh channel which can be used for receiving composite events of type `(Event [X Y])`; the composite event is generated each time when adjacent events `x` and `y` (or `y` and `x`) are received by mediator `m` on channels `c1` and `c2`. The composite events are assumed to be delivered to address `a`.

```
without = /[#X #Y ^(Event X) ^(Event Y) Address Address /^(Event X)]
```

`(without c1 c2 m a)` returns a new fresh channel which can be used for receiving composite events of type `(Event X)`; the composite event is generated by mediator `m` each time when event `x` occurs and is received on channel `c1` without event `y` having occurred first and being received on channel `c2`. The composite events are assumed to be delivered to address `a`.

```
followbywithout = /[#X #Y #Z ^(Event X) ^(Event Y) ^(Event Z) Address Address /^(Event [X Y])]
```

`(followbywithout c1 c2 c3 m a)` returns a new fresh channel which can be used for receiving composite events of type `(Event [X Y])`; the composite event is generated each time when events `x` and `y` are received by mediator `m` on channels `c1` and `c2` and the creation of event `y` follows `x` (in time) and there was no event of type `(Event Z)` generated between events `x` and `y` and received by mediator `m` on channel `c3`. The composite events are assumed to be delivered to address `a`.

```
followbywithout_ = /[#X #Y #Z ^(Event X) ^(Event Y) ^(Event Z) Address Address /^(Event [X Y])]
```

(followbywithout_ c1 c2 c3 m a) returns a new fresh channel which can be used for receiving composite events of type (Event [X Y]); the composite event is generated each time when event y is received by mediator m on channel c2 and there was some event x received by mediator m on channel c1 which was generated earlier than y (only the most recent x is considered) and there was no event of type (Event Z) generated between events x and y and received by mediator m on channel c3. The composite events are assumed to be delivered to address a.

```
_followbywithout = /[#X #Y #Z ^(Event X) ^(Event Y) ^(Event Z) Address Address /^(Event [X Y])]
```

(_followbywithout c1 c2 c3 m a) returns a new fresh channel which can be used for receiving composite events of type (Event [X Y]); each time when event y is received by mediator m on channel c2, the mediator generates one composite event for each event x received by mediator m on channel c1 which was generated earlier than y and not used for generating this type of composite events before, assuming that there was no event of type (Event Z) generated between events x and y and received by mediator m on channel c3. The composite events are assumed to be delivered to address a.

```
_followbywithout_= /[#X #Y #Z ^(Event X) ^(Event Y) ^(Event Z) Address Address /^(Event [X Y])]
```

(_followbywithout_ c1 c2 c3 m a) returns a new fresh channel which can be used for receiving composite events of type (Event [X Y]); the composite events generated are a summation of events which would be generated by _followbywithout and followbywithout_.

```
followby  = /[#X #Y ^(Event X) ^(Event Y) Address Address /^(Event [X Y])]
```

(followby c1 c2 m a) returns a new fresh channel which can be used for receiving composite events of type (Event [X Y]); the composite event is generated each time when events x and y are received by mediator m on channels c1 and c2 and the creation of event y follows x (in time). The composite events are assumed to be delivered to address a.

```
followby_ = /[#X #Y ^(Event X) ^(Event Y) Address Address /^(Event [X Y])]
```

(followby_ c1 c2 m a) returns a new fresh channel which can be used for receiving composite events of type (Event [X Y]); the composite event is generated each time when event y is received by mediator m on channel c2 and there was some event x received by mediator m on channel c1 which was generated earlier than y (only the most recent x is considered). The composite events are assumed to be delivered to address a.

```
_followby = /[#X #Y ^(Event X) ^(Event Y) Address Address /^(Event [X Y])]
```

(_followby c1 c2 m a) returns a new fresh channel which can be used for receiving composite events of type (Event [X Y]); each time when event y is received by mediator m on channel c2, the mediator generates one composite event for each event x received by mediator m on channel c1 which was generated earlier than y and not used for generating this type of composite events before. The composite events are assumed to be delivered to address a.

```
_followby_= /[#X #Y ^(Event X) ^(Event Y) Address Address /^(Event [X Y])]
```

(_followby_ c1 c2 m a) returns a new fresh channel which can be used for receiving composite events of type (Event [X Y]); the composite events generated are a summation of events which would be generated by _followby and followby_.

```
timedinclusiveor  = /[#X #Y ^(Event X) ^(Event Y) Time Address Address /^(Event [X Y])]
timedwithout = /[#X #Y ^(Event X) ^(Event Y) Time Address Address /^(Event X)]
timedfollowbywithout = /[#X #Y #Z ^(Event X) ^(Event Y) ^(Event Z) Time Address Address /^(Event [X Y])]
timedfollowbywithout_ = /[#X #Y #Z ^(Event X) ^(Event Y) ^(Event Z) Time Address Address /^(Event [X Y])]
_timedfollowbywithout = /[#X #Y #Z ^(Event X) ^(Event Y) ^(Event Z) Time Address Address /^(Event [X Y])]
```

```
_timedfollowbywithout_ = /[#X #Y #Z ^(Event X) ^(Event Y) ^(Event Z) Time Address Address /^(Event [X Y])]
timedfollowby  = /[#X #Y ^(Event X) ^(Event Y) Time Address Address /^(Event [X Y])]
timedfollowby_ = /[#X #Y ^(Event X) ^(Event Y) Time Address Address /^(Event [X Y])]
_timedfollowby = /[#X #Y ^(Event X) ^(Event Y) Time Address Address /^(Event [X Y])]
_timedfollowby_= /[#X #Y ^(Event X) ^(Event Y) Time Address Address /^(Event [X Y])]
```

Timed versions of the above. For example, `(followby c1 c2 t m a)` returns a new fresh channel which can be used for receiving event notifications from mediator `m` during a period of `t` seconds. After `t` seconds event notifications will be discarded at the mediator.

Chapter 5

Graphics Library

5.1 Graphics/Graphics: Machine-independent Graphics Primitives

This library implements an interface to the Objective Caml graphics library. The graphics library provides a set of portable drawing primitives. Drawing takes place in a separate window that is created when open_graph is called[1].

5.1.1 Types

```
#Status =
  [mouse_x = Int               {- X coordinate of the mouse -}
   mouse_y = Int               {- Y coordinate of the mouse -}
   button = Bool               {- true if a mouse button is pressed -}
   keypressed = Bool           {- true if a key has been pressed -}
   key = Char]                 {- the character for the key pressed -}
```

To report events.

```
#Event =
  [Button_down>[]              {- A mouse button is pressed -}
   Button_up>[]                {- A mouse button is released -}
   Key_pressed>[]              {- A key is pressed -}
   Mouse_motion>[]             {- The mouse is moved -}
   Poll>[]]                    {- Don't wait; return immediately -}
```

To specify events to wait for.

```
#Color
```

A color is specified by its r, g, b components. Each component is in the range 0..255. The three components are packed in an integer 0xRRGGBB, where RR are the two hexadecimal digits for the red component, GG for the green component, BB for the blue component.

5.1.2 Initializations

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

Show the graphics window or switch the screen to graphic mode. The graphics window is cleared and the current point is set to (0, 0). The string argument is used to pass optional information on the desired graphics mode, the graphics window size, and so on. Its interpretation is implementation-dependent. If the empty string is given, a sensible default is selected.

[1]Our documentation uses extracts from Graphics manual pages of "The Objective Caml system, release 2.04" by Xavier Leroy.

```
close_graph = /[Sig]
```

Delete the graphics window or switch the screen back to text mode.

```
clear_graph = /[Sig]
```

Erase the graphics window.

```
size_x = /[/Int]
size_y = /[/Int]
```

Return the size of the graphics window. Coordinates of the screen pixels range over `0` .. `(size_x)-1` and `0` .. `(size_y)-1`. Drawings outside of this rectangle are clipped, without causing an error. The origin `(0,0)` is at the lower left corner.

5.1.3 Colors

```
rgb = /[Int Int Int /Color]
```

`rgb r g b` returns the integer encoding the color with red component `r`, green component `g`, and blue omponent `b`. `r`, `g` and `b` are in the range `0..255`.

```
set_color = /[Color Sig]
```

Set the current drawing color.

```
black   = Color
white   = Color
red     = Color
green   = Color
blue    = Color
yellow  = Color
cyan    = Color
magenta = Color
```

Some predefined colors.

```
background = Color
foreground = Color
```

Default background and foreground colors (usually, either black foreground on a white background or white foreground on a black background). `clear_graph` fills the screen with the `background` color. The initial drawing color is `foreground`.

5.1.4 Point and line drawing

```
plot = /[Int Int Sig]
```

Plot the given point with the current drawing color.

```
point_color = /[Int Int /Color]
```

Return the color of the given point in the backing store (see "Double buffering" below)

```
moveto = /[Int Int Sig]
```

Position the current point.

```
rmoveto = /[Int Int Sig]
```

`rmoveto dx dy` translates the current point by the given vector.

```
current_point = /[/[Int Int]]
```

Return the position of the current point.

```
lineto = /[Int Int Sig]
```

Draw a line with endpoints the current point and the given point, and move the current point to the given point.

```
rlineto = /[Int Int Sig]
```

Draws a line with endpoints the current point and the current point translated by the given vector, and move the current point to this point.

```
draw_rect = /[Int Int Int Int Sig]
```

`draw_rect x y w h` draws the rectangle with lower left corner at `x`,`y`, width `w` and height `h`. The current point is unchanged.

```
draw_circle = /[Int Int Int Sig]
```

`draw_circle x y r` draws a circle with center `x`,`y` and radius `r`. The current point is unchanged.

```
set_line_width = /[Int Sig]
```

Set the width of points and lines drawn with the functions above. Under X Windows, `set_line_width 0` selects a width of `1` pixel and a faster, but less precise drawing algorithm than the one used when `set_line_width 1` is specified.

5.1.5 Text drawing

```
draw_char = /[Char Sig]
draw_string = /[String Sig]
```

Draw a character or a character string with lower left corner at current position. After drawing, the current position is set to the lower right corner of the text drawn.

```
set_font = /[String Sig]
set_text_size = /[Int Sig]
```

Set the font and character size used for drawing text. The interpretation of the arguments to `set_font` and `set_text_size` is implementation-dependent.

```
text_size = /[String /[Int Int]]
```

Return the dimensions of the given text, if it were drawn with the current font and size.

5.1.6 Filling

```
fill_rect = /[Int Int Int Int Sig]
```

`fill_rect x y w h` fills the rectangle with lower left corner at `x,y`, width `w` and height `h`, with the current color.

```
fill_circle = /[Int Int Int Sig]
```

Fill a circle with the current color. The parameters are the same as for `draw_circle`.

5.1.7 Mouse and keyboard events

```
wait_next_event = /[(List Event) /Status]
```

Wait until one of the events specified in the given event list occurs, and return the status of the mouse and keyboard at that time. If `Poll` is given in the event list, return immediately with the current status. If the mouse cursor is outside of the graphics window, the `mouse_x` and `mouse_y` fields of the event are outside the range `0..(size_x)-1`, `0..(size_y)-1`. Keypresses are queued, and dequeued one by one when the `Key_pressed` event is specified.

5.1.8 Mouse and keyboard polling

```
mouse_pos = /[/[Int Int]]
```

Return the position of the mouse cursor, relative to the graphics window. If the mouse cursor is outside of the graphics window, `(mouse_pos)` returns a point outside of the range `0..(size_x)-1`, `0..(size_y)-1`.

```
button_down = /[/Bool]
```

Return `true` if the mouse button is pressed, `false` otherwise.

```
read_key = /[/Char]
```

Wait for a key to be pressed, and return the corresponding character. Keypresses are queued.

```
key_pressed = /[/Bool]
```

Return `true` if a keypress is available; that is, if `read_key` would not block.

```
echo_keys = /[/String]
```

Wait for a key to be pressed and draw the key if not Enter; otherwise return the character string. After drawing, the current position is set to the lower right corner of the text drawn.

5.1.9 Sound

```
sound = /[Int Int Sig]
```

`(sound freq dur)` plays a sound at frequency `freq` (in hertz) for a duration `dur` (in miliseconds).

5.1.10 Double buffering

```
auto_synchronize = /[Bool Sig]
```

By default, drawing takes place both on the window displayed on screen, and in a memory area (the "backing store"). The backing store image is used to re-paint the on-screen window when necessary.

To avoid flicker during animations, it is possible to turn off on-screen drawing, perform a number of drawing operations in the backing store only, then refresh the on-screen window explicitly.

`auto_synchronize false` turns on-screen drawing off. All subsequent drawing commands are performed on the backing store only.

`auto_synchronize true` refreshes the on-screen window from the backing store (as per synchronize), then turns on-screen drawing back on. All subsequent drawing commands are performed both on screen and in the backing store.

The default drawing mode corresponds to `auto_synchronize true`.

```
synchronize = /[Sig]
```

Synchronize the backing store and the on-screen window, by copying the contents of the backing store onto the graphics window.

```
display_mode = /[Bool Sig]
```

Set display mode on or off. When turned on, drawings are done in the graphics window; when turned off, drawings do not affect the graphics window. This occurs independently of drawing into the backing store (see the function `remember_mode` below). Default display mode is on.

```
remember_mode = /[Bool Sig]
```

Set remember mode on or off. When turned on, drawings are done in the backing store; when turned off, the backing store is unaffected by drawings. This occurs independently of drawing onto the graphics window (see the function `display_mode` above). Default remember mode is on.

5.1.11 Abbreviations

```
val open_graph=graphics.open_graph
val close_graph=graphics.close_graph
val clear_graph=graphics.clear_graph
val size_x=graphics.size_x
val size_y=graphics.size_y
val rgb=graphics.rgb
val set_color=graphics.set_color
val black=graphics.black
val white=graphics.white
val red=graphics.red
val green=graphics.green
val blue=graphics.blue
val yellow=graphics.yellow
val cyan=graphics.cyan
val magenta=graphics.magenta
val background=graphics.background
val foreground=graphics.foreground
val plot=graphics.plot
val point_color=graphics.point_color
val moveto=graphics.moveto
val rmoveto=graphics.rmoveto
val current_point=graphics.current_point
val lineto=graphics.lineto
val rlineto=graphics.rlineto
val draw_rect=graphics.draw_rect
val draw_circle=graphics.draw_circle
val set_line_width=graphics.set_line_width
val draw_char=graphics.draw_char
val draw_string=graphics.draw_string
val set_font=graphics.set_font
val set_text_size=graphics.set_text_size
val text_size=graphics.text_size
val fill_rect=graphics.fill_rect
val fill_circle=graphics.fill_circle
val wait_next_event=graphics.wait_next_event
val mouse_pos=graphics.mouse_pos
val button_down=graphics.button_down
val read_key=graphics.read_key
val key_pressed=graphics.key_pressed
```

```
val echo_keys=graphics.echo_keys
val sound=graphics.sound
val auto_synchronize=graphics.auto_synchronize
val synchronize=graphics.synchronize
val display_mode=graphics.display_mode
val remember_mode=graphics.remember_mode
```

Bibliography

[PT97] Benjamin C. Pierce and David N. Turner. Pict libraries manual. Available electronically, 1997.

[Woj00] Paweł T. Wojciechowski. *The Nomadic Pict System*, 2000. Available electronically as part of the Nomadic Pict distribution.