Standard Libraries
for the
Haskell 98
Programming Language

1 February 1999

Simon Peyton Jones[8] [editor]
John Hughes[3] [editor]
Lennart Augustsson[3]
Dave Barton[7]
Brian Boutel[4]
Warren Burton[5]
Joseph Fasel[6]
Kevin Hammond[2]
Ralf Hinze[12]
Paul Hudak[1]
Thomas Johnsson[3]
Mark Jones[9]
John Launchbury[14]
Erik Meijer[10]
John Peterson[1]
Alastair Reid[1]
Colin Runciman[13]
Philip Wadler[11]

Authors' affiliations: (1) Yale University (2) University of St. Andrews (3) Chalmers University of Technology (4) Victoria University of Wellington (5) Simon Fraser University (6) Los Alamos National Laboratory (7) Intermetrics (8) Microsoft Research, Cambridge (9) University of Nottingham (10) Utrecht University (11) Bell Labs (12) University of Bonn (13) York University (14) Oregon Graduate Institute

Contents

Preface

This document defines the standard libraries for Haskell 98.

The libraries presented here represent a selection of basic functionality that is expected to be useful to many Haskell programmers. Most implementations provide further libraries which are not a recognized part of the Haskell standard.

The latest version of this report, as well many other available libraries, can be found on the web at `http://haskell.org`.

We would like to express our thanks to those who have contributed directly or indirectly to this report without being named as authors, including Olaf Chitil, Tony Davie, Sigbjorn Finne, Andy Gill, Mike Gunter, Fergus Henderson, Kent Karlsson, Sandra Loosemore, Graeme Moss, Sven Panne, Keith Wansbrough.

1 Introduction

This document defines the standard libraries for Haskell 98. Like the `Prelude`, these libraries are a required part of a Haskell implementation. Unlike the Prelude, however, these modules must be *explicitly* imported into scope.

When possible, library functions are described solely by executable Haskell code. Functions which require implementation-dependent primitives are represented by type signatures without definitions. Some data types are implementation-dependent: these are indicated by comments in the source.

The code found here is a *specification*, rather than an *implementation*. Implementations may choose more efficient versions of these functions. However, all properties of these specifications must be preserved, including strictness properties.

Classes defined in libraries may be derivable. This report includes the derivation of such classes when appropriate. When `Prelude` types are instances of derivable library classes a commented empty instance declaration is used. The comment, "as derived", indicates that the instance is the same as would have been generated by a `deriving` in the Prelude type declaration.

The following table summarises the fixities of all the operators introduced by the standard libraries:

Precedence	Left associative operators	Non-associative operators	Right associative operators
9	`Array.!`, `Array.//`		
7	`Ratio.%`		
6		`Complex.:+`	
5		`List.\\`	

Table 1: Precedences and fixities of library operators

2 Rational Numbers

```
module  Ratio (
    Ratio, Rational, (%), numerator, denominator, approxRational ) where

infixl 7  %
data  (Integral a)      => Ratio a = ...
type  Rational          =  Ratio Integer
(%)                     :: (Integral a) => a -> a -> Ratio a
numerator, denominator  :: (Integral a) => Ratio a -> a
approxRational          :: (RealFrac a) => a -> a -> Rational
instance  (Integral a)  => Eq         (Ratio a)  where ...
instance  (Integral a)  => Ord        (Ratio a)  where ...
instance  (Integral a)  => Num        (Ratio a)  where ...
instance  (Integral a)  => Real       (Ratio a)  where ...
instance  (Integral a)  => Fractional (Ratio a)  where ...
instance  (Integral a)  => RealFrac   (Ratio a)  where ...
instance  (Integral a)  => Enum       (Ratio a)  where ...
instance  (Read a,Integral a) => Read (Ratio a)  where ...
instance  (Integral a)  => Show       (Ratio a)  where ...
```

For each `Integral` type t, there is a type `Ratio` t of rational pairs with components of type t. The type name `Rational` is a synonym for `Ratio Integer`.

`Ratio` is an instance of classes `Eq`, `Ord`, `Num`, `Real`, `Fractional`, `RealFrac`, `Enum`, `Read`, and `Show`. In each case, the instance for `Ratio` t simply "lifts" the corresponding operations over t. If t is a bounded type, the results may be unpredictable; for example `Ratio Int` may give rise to integer overflow even for rational numbers of small absolute size.

The operator `(%)` forms the ratio of two integral numbers, reducing the fraction to terms with no common factor and such that the denominator is positive. The functions `numerator` and `denominator` extract the components of a ratio; these are in reduced form with a positive denominator. `Ratio` is an abstract type. For example, `12 % 8` is reduced to 3/2 and `12 % (-8)` is reduced to (-3)/2.

The `approxRational` function, applied to two real fractional numbers `x` and `epsilon`, returns the simplest rational number within the open interval (`x` − `epsilon`, `x` + `epsilon`). A rational number n/d in reduced form is said to be simpler than another n'/d' if $|n| \leq |n'|$ and $d \leq d'$. Note that it can be proved that any real interval contains a unique simplest rational.

2.1 Library `Ratio`

```
-- Standard functions on rational numbers

module  Ratio (
    Ratio, Rational, (%), numerator, denominator, approxRational ) where

infixl 7  %

prec = 7 :: Int

data  (Integral a)      => Ratio a = !a :% !a  deriving (Eq)
type  Rational          =  Ratio Integer

(%)                     :: (Integral a) => a -> a -> Ratio a
numerator, denominator  :: (Integral a) => Ratio a -> a
approxRational          :: (RealFrac a) => a -> a -> Rational

-- "reduce" is a subsidiary function used only in this module.
-- It normalises a ratio by dividing both numerator
-- and denominator by their greatest common divisor.
--
-- E.g., 12 `reduce` 8    ==  3 :%   2
--       12 `reduce` (-8) ==  3 :% (-2)

reduce _ 0              =  error "Ratio.% : zero denominator"
reduce x y              =  (x `quot` d) :% (y `quot` d)
                           where d = gcd x y

x % y                   =  reduce (x * signum y) (abs y)

numerator (x :% _)      =  x

denominator (_ :% y)    =  y

instance  (Integral a)  => Ord (Ratio a)  where
    (x:%y) <= (x':%y')  =  x * y' <= x' * y
    (x:%y) <  (x':%y')  =  x * y' <  x' * y

instance  (Integral a)  => Num (Ratio a)  where
    (x:%y) + (x':%y')   =  reduce (x*y' + x'*y) (y*y')
    (x:%y) * (x':%y')   =  reduce (x * x') (y * y')
    negate (x:%y)       =  (-x) :% y
    abs (x:%y)          =  abs x :% y
    signum (x:%y)       =  signum x :% 1
    fromInteger x       =  fromInteger x :% 1

instance  (Integral a)  => Real (Ratio a)  where
    toRational (x:%y)   =  toInteger x :% toInteger y
```

```
instance  (Integral a)  => Fractional (Ratio a)  where
    (x:%y) / (x':%y')   =  (x*y') % (y*x')
    recip (x:%y)        =  if x < 0 then (-y) :% (-x) else y :% x
    fromRational (x:%y) =  fromInteger x :% fromInteger y

instance  (Integral a)  => RealFrac (Ratio a)  where
    properFraction (x:%y) = (fromIntegral q, r:%y)
                            where (q,r) = quotRem x y

instance  (Integral a)  => Enum (Ratio a)  where
    toEnum           =  fromIntegral
    fromEnum         =  fromInteger . truncate  -- May overflow
    enumFrom         =  numericEnumFrom         -- These numericEnumXXX functions
    enumFromThen     =  numericEnumFromThen     -- are as defined in Prelude.hs
    enumFromTo       =  numericEnumFromTo       -- but not exported from it!
    enumFromThenTo   =  numericEnumFromThenTo

instance  (Read a, Integral a)  => Read (Ratio a)  where
    readsPrec p  =  readParen (p > prec)
                             (\r -> [(x%y,u) | (x,s)   <- reads r,
                                               ("%",t) <- lex s,
                                               (y,u)   <- reads t ])

instance  (Integral a)  => Show (Ratio a)  where
    showsPrec p (x:%y)  =  showParen (p > prec)
                              (shows x . showString " % " . shows y)

approxRational x eps    =  simplest (x-eps) (x+eps)
        where simplest x y | y < x      =  simplest y x
                           | x == y     =  xr
                           | x > 0      =  simplest' n d n' d'
                           | y < 0      =  - simplest' (-n') d' (-n) d
                           | otherwise  =  0 :% 1
                                        where xr@(n:%d) = toRational x
                                              (n':%d')  = toRational y

              simplest' n d n' d'       -- assumes 0 < n%d < n'%d'
                        | r == 0     =  q :% 1
                        | q /= q'    =  (q+1) :% 1
                        | otherwise  =  (q*n''+d'') :% n''
                                     where (q,r)      =  quotRem n d
                                           (q',r')    =  quotRem n' d'
                                           (n'':%d'') =  simplest' d' r' d r
```

3 Complex Numbers

```
module  Complex (
    Complex((:+)), realPart, imagPart, conjugate,
    mkPolar, cis, polar, magnitude, phase ) where

infix  6  :+

data  (RealFloat a)     => Complex a = !a :+ !a

realPart, imagPart      :: (RealFloat a) => Complex a -> a
conjugate               :: (RealFloat a) => Complex a -> Complex a
mkPolar                 :: (RealFloat a) => a -> a -> Complex a
cis                     :: (RealFloat a) => a -> Complex a
polar                   :: (RealFloat a) => Complex a -> (a,a)
magnitude, phase        :: (RealFloat a) => Complex a -> a

instance  (RealFloat a) => Eq         (Complex a)  where ...
instance  (RealFloat a) => Read       (Complex a)  where ...
instance  (RealFloat a) => Show       (Complex a)  where ...
instance  (RealFloat a) => Num        (Complex a)  where ...
instance  (RealFloat a) => Fractional (Complex a)  where ...
instance  (RealFloat a) => Floating   (Complex a)  where ...
```

Complex numbers are an algebraic type. The constructor `(:+)` forms a complex number from its real and imaginary rectangular components. This constructor is strict: if either the real part or the imaginary part of the number is $\perp$, the entire number is $\perp$. A complex number may also be formed from polar components of magnitude and phase by the function `mkPolar`. The function `cis` produces a complex number from an angle t. Put another way, `cis` t is a complex value with magnitude *1* and phase t (modulo 2π).

The function `polar` takes a complex number and returns a (magnitude, phase) pair in canonical form: The magnitude is nonnegative, and the phase, in the range $(-\pi, \pi]$; if the magnitude is zero, then so is the phase.

The functions `realPart` and `imagPart` extract the rectangular components of a complex number and the functions `magnitude` and `phase` extract the polar components of a complex number. The function `conjugate` computes the conjugate of a complex number in the usual way.

The magnitude and sign of a complex number are defined as follows:

```
abs z             =  magnitude z :+ 0
signum 0          =  0
signum z@(x:+y)   =  x/r :+ y/r  where r = magnitude z
```

That is, `abs` z is a number with the magnitude of z, but oriented in the positive real direction, whereas `signum` z has the phase of z, but unit magnitude.

3.1 Library Complex

```
module Complex(Complex((:+)), realPart, imagPart, conjugate, mkPolar,
               cis, polar, magnitude, phase)  where

infix  6  :+

data  (RealFloat a)     => Complex a = !a :+ !a  deriving (Eq,Read,Show)

realPart, imagPart :: (RealFloat a) => Complex a -> a
realPart (x:+y)  =  x
imagPart (x:+y)  =  y

conjugate        :: (RealFloat a) => Complex a -> Complex a
conjugate (x:+y) =  x :+ (-y)

mkPolar          :: (RealFloat a) => a -> a -> Complex a
mkPolar r theta  =  r * cos theta :+ r * sin theta

cis              :: (RealFloat a) => a -> Complex a
cis theta        =  cos theta :+ sin theta

polar            :: (RealFloat a) => Complex a -> (a,a)
polar z          =  (magnitude z, phase z)

magnitude :: (RealFloat a) => Complex a -> a
magnitude (x:+y) =  scaleFloat k
                     (sqrt ((scaleFloat mk x)^2 + (scaleFloat mk y)^2))
                    where k  = max (exponent x) (exponent y)
                          mk = - k

phase :: (RealFloat a) => Complex a -> a
phase (0 :+ 0) = 0
phase (x :+ y) = atan2 y x

instance  (RealFloat a) => Num (Complex a)  where
    (x:+y) + (x':+y')   =  (x+x') :+ (y+y')
    (x:+y) - (x':+y')   =  (x-x') :+ (y-y')
    (x:+y) * (x':+y')   =  (x*x'-y*y') :+ (x*y'+y*x')
    negate (x:+y)       =  negate x :+ negate y
    abs z               =  magnitude z :+ 0
    signum 0            =  0
    signum z@(x:+y)     =  x/r :+ y/r  where r = magnitude z
    fromInteger n       =  fromInteger n :+ 0
```

```
instance  (RealFloat a) => Fractional (Complex a)  where
    (x:+y) / (x':+y')   =  (x*x''+y*y'') / d :+ (y*x''-x*y'') / d
                           where x'' = scaleFloat k x'
                                 y'' = scaleFloat k y'
                                 k   = - max (exponent x') (exponent y')
                                 d   = x'*x'' + y'*y''

    fromRational a      =  fromRational a :+ 0

instance  (RealFloat a) => Floating (Complex a) where
    pi             =  pi :+ 0
    exp (x:+y)     =  expx * cos y :+ expx * sin y
                      where expx = exp x
    log z          =  log (magnitude z) :+ phase z

    sqrt 0         =  0
    sqrt z@(x:+y)  =  u :+ (if y < 0 then -v else v)
                      where (u,v) = if x < 0 then (v',u') else (u',v')
                            v'    = abs y / (u'*2)
                            u'    = sqrt ((magnitude z + abs x) / 2)

    sin (x:+y)     =  sin x * cosh y :+ cos x * sinh y
    cos (x:+y)     =  cos x * cosh y :+ (- sin x * sinh y)
    tan (x:+y)     =  (sinx*coshy:+cosx*sinhy)/(cosx*coshy:+(-sinx*sinhy))
                      where sinx  = sin x
                            cosx  = cos x
                            sinhy = sinh y
                            coshy = cosh y

    sinh (x:+y)    =  cos y * sinh x :+ sin  y * cosh x
    cosh (x:+y)    =  cos y * cosh x :+ sin y * sinh x
    tanh (x:+y)    =  (cosy*sinhx:+siny*coshx)/(cosy*coshx:+siny*sinhx)
                      where siny  = sin y
                            cosy  = cos y
                            sinhx = sinh x
                            coshx = cosh x

    asin z@(x:+y)  =  y':+(-x')
                      where  (x':+y') = log (((-y):+x) + sqrt (1 - z*z))
    acos z@(x:+y)  =  y'':+(-x'')
                      where (x'':+y'') = log (z + ((-y'):+x'))
                            (x':+y')   = sqrt (1 - z*z)
    atan z@(x:+y)  =  y':+(-x')
                      where (x':+y') = log (((1-y):+x) / sqrt (1+z*z))

    asinh z        =  log (z + sqrt (1+z*z))
    acosh z        =  log (z + (z+1) * sqrt ((z-1)/(z+1)))
    atanh z        =  log ((1+z) / sqrt (1-z*z))
```

4 Numeric

```
module Numeric(fromRat,
               showSigned, showInt,
               readSigned, readInt,
               readDec, readOct, readHex,
               floatToDigits,
               showEFloat, showFFloat, showGFloat, showFloat,
               readFloat, lexDigits) where

fromRat        :: (RealFloat a) => Rational -> a
showSigned     :: (Real a) => (a -> ShowS) -> Int -> a -> ShowS
showInt        :: Integral a => a -> ShowS
readSigned     :: (Real a) => ReadS a -> ReadS a
readInt        :: (Integral a) =>
                    a -> (Char -> Bool) -> (Char -> Int) -> ReadS a
readDec        :: (Integral a) => ReadS a
readOct        :: (Integral a) => ReadS a
readHex        :: (Integral a) => ReadS a

showEFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showFFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showGFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showFloat      :: (RealFloat a) => a -> ShowS

floatToDigits  :: (RealFloat a) => Integer -> a -> ([Int], Int)

readFloat      :: (RealFloat a) => ReadS a
lexDigits      :: ReadS String
```

This library contains assorted numeric functions, many of which are used in the standard Prelude. Most are self-explanatory. The `floatToDigits` function converts a floating point value into a series of digits and an exponent of a selected base. This is used to build a set of floating point formatting functions.

4.1 Library `Numeric`

```
module Numeric(fromRat,
               showSigned, showInt,
               readSigned, readInt,
               readDec, readOct, readHex,
               floatToDigits,
               showEFloat, showFFloat, showGFloat, showFloat,
               readFloat, lexDigits) where
```

```
import Char
import Ratio
import Array

-- This converts a rational to a floating.  This should be used in the
-- Fractional instances of Float and Double.

fromRat :: (RealFloat a) => Rational -> a
fromRat x =
    if x == 0 then encodeFloat 0 0              -- Handle exceptional cases
    else if x < 0 then - fromRat' (-x)          -- first.
    else fromRat' x

-- Conversion process:
-- Scale the rational number by the RealFloat base until
-- it lies in the range of the mantissa (as used by decodeFloat/encodeFloat).
-- Then round the rational to an Integer and encode it with the exponent
-- that we got from the scaling.
-- To speed up the scaling process we compute the log2 of the number to get
-- a first guess of the exponent.
fromRat' :: (RealFloat a) => Rational -> a
fromRat' x = r
  where b = floatRadix r
        p = floatDigits r
        (minExp0, _) = floatRange r
        minExp = minExp0 - p            -- the real minimum exponent
        xMin = toRational (expt b (p-1))
        xMax = toRational (expt b p)
        p0 = (integerLogBase b (numerator x) -
              integerLogBase b (denominator x) - p) `max` minExp
        f = if p0 < 0 then 1 % expt b (-p0) else expt b p0 % 1
        (x', p') = scaleRat (toRational b) minExp xMin xMax p0 (x / f)
        r = encodeFloat (round x') p'

-- Scale x until xMin <= x < xMax, or p (the exponent) <= minExp.
scaleRat :: Rational -> Int -> Rational -> Rational ->
             Int -> Rational -> (Rational, Int)
scaleRat b minExp xMin xMax p x =
    if p <= minExp then
        (x, p)
    else if x >= xMax then
        scaleRat b minExp xMin xMax (p+1) (x/b)
    else if x < xMin  then
        scaleRat b minExp xMin xMax (p-1) (x*b)
    else
        (x, p)
```

```
-- Exponentiation with a cache for the most common numbers.
minExpt = 0::Int
maxExpt = 1100::Int
expt :: Integer -> Int -> Integer
expt base n =
    if base == 2 && n >= minExpt && n <= maxExpt then
        expts!n
    else
        base^n

expts :: Array Int Integer
expts = array (minExpt,maxExpt) [(n,2^n) | n <- [minExpt .. maxExpt]]

-- Compute the (floor of the) log of i in base b.
-- Simplest way would be just divide i by b until it's smaller then b,
-- but that would be very slow!  We are just slightly more clever.
integerLogBase :: Integer -> Integer -> Int
integerLogBase b i =
     if i < b then
        0
     else
        -- Try squaring the base first to cut down the number of divisions.
        let l = 2 * integerLogBase (b*b) i
            doDiv :: Integer -> Int -> Int
            doDiv i l = if i < b then l else doDiv (i `div` b) (l+1)
        in  doDiv (i `div` (b^l)) l

-- Misc utilities to show integers and floats

showSigned    :: Real a => (a -> ShowS) -> Int -> a -> ShowS
showSigned showPos p x | x < 0 = showParen (p > 6)
                                           (showChar '-' . showPos (-x))
                       | otherwise = showPos x

-- showInt is used for positive numbers only
showInt    :: Integral a => a -> ShowS
showInt n r | n < 0 = error "Numeric.showInt: can't show negative numbers"
            | otherwise =
              let (n',d) = quotRem n 10
                  r'     = toEnum (fromEnum '0' + fromIntegral d) : r
              in  if n' == 0 then r' else showInt n' r'
```

```
readSigned :: (Real a) => ReadS a -> ReadS a
readSigned readPos = readParen False read'
                     where read' r  = read'' r ++
                                      [(-x,t) | ("-",s) <- lex r,
                                                (x,t)   <- read'' s]
                           read'' r = [(n,s)  | (str,s) <- lex r,
                                                (n,"")  <- readPos str]

-- readInt reads a string of digits using an arbitrary base.
-- Leading minus signs must be handled elsewhere.

readInt :: (Integral a) => a -> (Char -> Bool) -> (Char -> Int) -> ReadS a
readInt radix isDig digToInt s =
   [(foldl1 (\n d -> n * radix + d) (map (fromIntegral . digToInt) ds), r)
          | (ds,r) <- nonnull isDig s ]

-- Unsigned readers for various bases
readDec, readOct, readHex :: (Integral a) => ReadS a
readDec = readInt 10 isDigit digitToInt
readOct = readInt  8 isOctDigit digitToInt
readHex = readInt 16 isHexDigit digitToInt

showEFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showFFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showGFloat     :: (RealFloat a) => Maybe Int -> a -> ShowS
showFloat      :: (RealFloat a) => a -> ShowS

showEFloat d x =  showString (formatRealFloat FFExponent d x)
showFFloat d x =  showString (formatRealFloat FFFixed d x)
showGFloat d x =  showString (formatRealFloat FFGeneric d x)
showFloat      =  showGFloat Nothing

-- These are the format types.  This type is not exported.

data FFormat = FFExponent | FFFixed | FFGeneric
```

```
formatRealFloat :: (RealFloat a) => FFFormat -> Maybe Int -> a -> String
formatRealFloat fmt decs x = s
  where base = 10
        s = if isNaN x then
                "NaN"
            else if isInfinite x then
                if x < 0 then "-Infinity" else "Infinity"
            else if x < 0 || isNegativeZero x then
                '-' : doFmt fmt (floatToDigits (toInteger base) (-x))
            else
                doFmt fmt (floatToDigits (toInteger base) x)
        doFmt fmt (is, e) =
            let ds = map intToDigit is
            in  case fmt of
                FFGeneric ->
                    doFmt (if e < 0 || e > 7 then FFExponent else FFFixed)
                          (is, e)
                FFExponent ->
                    case decs of
                    Nothing ->
                        case ds of
                         ['0'] -> "0.0e0"
                         [d]   -> d : ".0e" ++ show (e-1)
                         d:ds  -> d : '.' : ds ++ 'e':show (e-1)
                    Just dec ->
                        let dec' = max dec 1 in
                        case is of
                         [0] -> '0':'.':take dec' (repeat '0') ++ "e0"
                         _ ->
                          let (ei, is') = roundTo base (dec'+1) is
                              d:ds = map intToDigit
                                         (if ei > 0 then init is' else is')
                          in d:'.':ds  ++ "e" ++ show (e-1+ei)
                FFFixed ->
                    case decs of
                    Nothing ->
                        let f 0 s ds = mk0 s ++ "." ++ mk0 ds
                            f n s "" = f (n-1) (s++"0") ""
                            f n s (d:ds) = f (n-1) (s++[d]) ds
                            mk0 "" = "0"
                            mk0 s = s
                        in  f e "" ds
                    Just dec ->
                        let dec' = max dec 0 in
```

```
                        if e >= 0 then
                            let (ei, is') = roundTo base (dec' + e) is
                                (ls, rs) = splitAt (e+ei) (map intToDigit is')
                            in  (if null ls then "0" else ls) ++
                                (if null rs then "" else '.' : rs)
                        else
                            let (ei, is') = roundTo base dec'
                                              (replicate (-e) 0 ++ is)
                                d : ds = map intToDigit
                                            (if ei > 0 then is' else 0:is')
                            in  d : '.' : ds

roundTo :: Int -> Int -> [Int] -> (Int, [Int])
roundTo base d is = case f d is of
                (0, is) -> (0, is)
                (1, is) -> (1, 1 : is)
  where b2 = base `div` 2
        f n [] = (0, replicate n 0)
        f 0 (i:_) = (if i >= b2 then 1 else 0, [])
        f d (i:is) =
            let (c, ds) = f (d-1) is
                i' = c + i
            in  if i' == base then (1, 0:ds) else (0, i':ds)

--
-- Based on "Printing Floating-Point Numbers Quickly and Accurately"
-- by R.G. Burger and R. K. Dybvig, in PLDI 96.
-- This version uses a much slower logarithm estimator.  It should be improved.

-- This function returns a list of digits (Ints in [0..base-1]) and an
-- exponent.

floatToDigits :: (RealFloat a) => Integer -> a -> ([Int], Int)
```

```
floatToDigits _ 0 = ([0], 0)
floatToDigits base x =
    let (f0, e0) = decodeFloat x
        (minExp0, _) = floatRange x
        p = floatDigits x
        b = floatRadix x
        minExp = minExp0 - p            -- the real minimum exponent
        -- Haskell requires that f be adjusted so denormalized numbers
        -- will have an impossibly low exponent.  Adjust for this.
        (f, e) = let n = minExp - e0
                 in  if n > 0 then (f0 `div` (b^n), e0+n) else (f0, e0)

        (r, s, mUp, mDn) =
           if e >= 0 then
               let be = b^e in
               if f == b^(p-1) then
                   (f*be*b*2, 2*b, be*b, b)
               else
                   (f*be*2, 2, be, be)
           else
               if e > minExp && f == b^(p-1) then
                   (f*b*2, b^(-e+1)*2, b, 1)
               else
                   (f*2, b^(-e)*2, 1, 1)
        k =
            let k0 =
                    if b==2 && base==10 then
                        -- logBase 10 2 is slightly bigger than 3/10 so
                        -- the following will err on the low side.  Ignoring
                        -- the fraction will make it err even more.
                        -- Haskell promises that p-1 <= logBase b f < p.
                        (p - 1 + e0) * 3 `div` 10
                    else
                        ceiling ((log (fromInteger (f+1)) +
                                 fromInt e * log (fromInteger b)) /
                                  log (fromInteger base))
                fixup n =
                    if n >= 0 then
                        if r + mUp <= expt base n * s then n else fixup (n+1)
                    else
                        if expt base (-n) * (r + mUp) <= s then n
                                                           else fixup (n+1)
            in  fixup k0

        gen ds rn sN mUpN mDnN =
```

```
            let (dn, rn') = (rn * base) `divMod` sN
                mUpN' = mUpN * base
                mDnN' = mDnN * base
            in  case (rn' < mDnN', rn' + mUpN' > sN) of
                (True,  False) -> dn : ds
                (False, True)  -> dn+1 : ds
                (True,  True)  -> if rn' * 2 < sN then dn : ds else dn+1 : ds
                (False, False) -> gen (dn:ds) rn' sN mUpN' mDnN'
        rds =
            if k >= 0 then
                gen [] r (s * expt base k) mUp mDn
            else
                let bk = expt base (-k)
                in  gen [] (r * bk) s (mUp * bk) (mDn * bk)
    in  (map toInt (reverse rds), k)

-- This floating point reader uses a less restrictive syntax for floating
-- point than the Haskell lexer.  The `.' is optional.

readFloat     :: (RealFloat a) => ReadS a
readFloat r    = [(fromRational ((n%1)*10^^(k-d)),t) | (n,d,s) <- readFix r,
                                                       (k,t)   <- readExp s]
                 where readFix r = [(read (ds++ds'), length ds', t)
                                   | (ds,d) <- lexDigits r,
                                     (ds',t) <- lexFrac d ]

                       lexFrac ('.':ds) = lexDigits ds
                       lexFrac s        = [("",s)]

                       readExp (e:s) | e `elem` "eE" = readExp' s
                       readExp s                     = [(0,s)]

                       readExp' ('-':s) = [(-k,t) | (k,t) <- readDec s]
                       readExp' ('+':s) = readDec s
                       readExp' s       = readDec s

lexDigits        :: ReadS String
lexDigits        =  nonnull isDigit

nonnull          :: (Char -> Bool) -> ReadS String
nonnull p s      =  [(cs,t) | (cs@(_:_),t) <- [span p s]]
```

5 Indexing Operations

```
module Ix ( Ix(range, index, inRange), rangeSize ) where

class  (Ord a) => Ix a  where
    range              :: (a,a) -> [a]
    index              :: (a,a) -> a -> Int
    inRange            :: (a,a) -> a -> Bool

rangeSize              :: (Ix a) => (a,a) -> Int

instance                  Ix Char      where ...
instance                  Ix Int       where ...
instance                  Ix Integer   where ...
instance  (Ix a, Ix b)  => Ix (a,b)     where ...
-- et cetera
instance                  Ix Bool      where ...
instance                  Ix Ordering  where ...
```

The `Ix` class is used to map a continuous subrange of values in a type onto integers. It is used primarily for array indexing (see Section 6). The `Ix` class contains the methods `range`, `index`, and `inRange`. The `index` operation maps a bounding pair, which defines the lower and upper bounds of the range, and a subscript, to an integer. The `range` operation enumerates all subscripts; the `inRange` operation tells whether a particular subscript lies in the range defined by a bounding pair.

An implementation is entitled to assume the following laws about these operations:

```
range (l,u) !! index (l,u) i == i   -- when i is in range
inRange (l,u) i == i `elem` range (l,u)
```

5.1 Deriving Instances of `Ix`

Derived instance declarations for the class `Ix` are only possible for enumerations (i.e. datatypes having only nullary constructors) and single-constructor datatypes, including arbitrarily large tuples, whose constituent types are instances of `Ix`.

- For an *enumeration*, the nullary constructors are assumed to be numbered left-to-right with the indices being 0 to $n-1$ inclusive. This is the same numbering defined by the `Enum` class. For example, given the datatype:

  ```
  data Colour = Red | Orange | Yellow | Green | Blue | Indigo | Violet
  ```

  we would have:

```
instance  (Ix a, Ix b)  => Ix (a,b) where
        range ((l,l'),(u,u'))
                = [(i,i') | i <- range (l,u), i' <- range (l',u')]
        index ((l,l'),(u,u')) (i,i')
                =  index (l,u) i * rangeSize (l',u') + index (l',u') i'
        inRange ((l,l'),(u,u')) (i,i')
                = inRange (l,u) i && inRange (l',u') i'

-- Instances for other tuples are obtained from this scheme:
--
--  instance  (Ix a1, Ix a2, ... , Ix ak) => Ix (a1,a2,...,ak)  where
--      range ((l1,l2,...,lk),(u1,u2,...,uk)) =
--          [(i1,i2,...,ik) | i1 <- range (l1,u1),
--                            i2 <- range (l2,u2),
--                            ...
--                            ik <- range (lk,uk)]
--
--      index ((l1,l2,...,lk),(u1,u2,...,uk)) (i1,i2,...,ik) =
--        index (lk,uk) ik + rangeSize (lk,uk) * (
--         index (lk-1,uk-1) ik-1 + rangeSize (lk-1,uk-1) * (
--          ...
--           index (l1,u1)))
--
--      inRange ((l1,l2,...lk),(u1,u2,...,uk)) (i1,i2,...,ik) =
--          inRange (l1,u1) i1 && inRange (l2,u2) i2 &&
--              ... && inRange (lk,uk) ik
```

Figure 1: Derivation of Ix instances

```
range   (Yellow,Blue)        ==  [Yellow,Green,Blue]
index   (Yellow,Blue) Green  ==  1
inRange (Yellow,Blue) Red    ==  False
```

- For *single-constructor datatypes*, the derived instance declarations are as shown for tuples in Figure 1.

5.2 Library Ix

```
module Ix ( Ix(range, index, inRange), rangeSize ) where

class  (Ord a) => Ix a  where
    range               :: (a,a) -> [a]
    index               :: (a,a) -> a -> Int
    inRange             :: (a,a) -> a -> Bool

rangeSize :: Ix a => (a,a) -> Int
rangeSize b@(l,h) | null (range b) = 0
                  | otherwise      = index b h + 1
        -- NB: replacing "null (range b)" by  "l > h" fails if
        -- the bounds are tuples.  For example,
        --      (2,1) > (1,2),
        -- but
        --      range ((2,1),(1,2)) = []

instance  Ix Char  where
    range (m,n)         = [m..n]
    index b@(c,c') ci
        | inRange b ci  =  fromEnum ci - fromEnum c
        | otherwise     =  error "Ix.index: Index out of range."
    inRange (c,c') i    =  c <= i && i <= c'

instance  Ix Int  where
    range (m,n)         = [m..n]
    index b@(m,n) i
        | inRange b i   =  i - m
        | otherwise     =  error "Ix.index: Index out of range."
    inRange (m,n) i     =  m <= i && i <= n

instance  Ix Integer  where
    range (m,n)         = [m..n]
    index b@(m,n) i
        | inRange b i   =  fromInteger (i - m)
        | otherwise     =  error "Ix.index: Index out of range."
    inRange (m,n) i     =  m <= i && i <= n

instance (Ix a,Ix b) => Ix (a, b) -- as derived, for all tuples
instance Ix Bool                  -- as derived
instance Ix Ordering              -- as derived
instance Ix ()                    -- as derived
```

6 Arrays

```
module  Array (
        module Ix,  -- export all of Ix for convenience
        Array, array, listArray, (!), bounds, indices, elems, assocs,
        accumArray, (//), accum, ixmap ) where

import Ix

infixl 9  !, //

data  (Ix a)    => Array a b = ...      -- Abstract

array           :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
listArray       :: (Ix a) => (a,a) -> [b] -> Array a b
(!)             :: (Ix a) => Array a b -> a -> b
bounds          :: (Ix a) => Array a b -> (a,a)
indices         :: (Ix a) => Array a b -> [a]
elems           :: (Ix a) => Array a b -> [b]
assocs          :: (Ix a) => Array a b -> [(a,b)]
accumArray      :: (Ix a) => (b -> c -> b) -> b -> (a,a) -> [(a,c)]
                             -> Array a b
(//)            :: (Ix a) => Array a b -> [(a,b)] -> Array a b
accum           :: (Ix a) => (b -> c -> b) -> Array a b -> [(a,c)]
                             -> Array a b
ixmap           :: (Ix a, Ix b) => (a,a) -> (a -> b) -> Array b c
                             -> Array a c

instance                            Functor (Array a) where ...
instance  (Ix a, Eq b)           => Eq   (Array a b)  where ...
instance  (Ix a, Ord b)          => Ord  (Array a b)  where ...
instance  (Ix a, Show a, Show b) => Show (Array a b)  where ...
instance  (Ix a, Read a, Read b) => Read (Array a b)  where ...
```

Haskell provides indexable *arrays*, which may be thought of as functions whose domains are isomorphic to contiguous subsets of the integers. Functions restricted in this way can be implemented efficiently; in particular, a programmer may reasonably expect rapid access to the components. To ensure the possibility of such an implementation, arrays are treated as data, not as general functions.

Since most array functions involve the class `Ix`, this module is exported from `Array` so that modules need not import both `Array` and `Ix`.

```
-- Scaling an array of numbers by a given number:
scale :: (Num a, Ix b) => a -> Array b a -> Array b a
scale x a = array b [(i, a!i * x) | i <- range b]
            where b = bounds a

-- Inverting an array that holds a permutation of its indices
invPerm :: (Ix a) => Array a a -> Array a a
invPerm a = array b [(a!i, i) | i <- range b]
            where b = bounds a

-- The inner product of two vectors
inner :: (Ix a, Num b) => Array a b -> Array a b -> b
inner v w = if b == bounds w
                then sum [v!i * w!i | i <- range b]
                else error "inconformable arrays for inner product"
            where b = bounds v
```

Figure 2: Array examples

6.1 Array Construction

If `a` is an index type and `b` is any type, the type of arrays with indices in `a` and elements in `b` is written `Array a b`. An array may be created by the function `array`. The first argument of `array` is a pair of *bounds*, each of the index type of the array. These bounds are the lowest and highest indices in the array, in that order. For example, a one-origin vector of length `10` has bounds `(1,10)`, and a one-origin `10` by `10` matrix has bounds `((1,1),(10,10))`.

The second argument of `array` is a list of *associations* of the form (*index*, *value*). Typically, this list will be expressed as a comprehension. An association `(i, x)` defines the value of the array at index `i` to be `x`. The array is undefined (i.e. $\perp$) if any index in the list is out of bounds. If any two associations in the list have the same index, the value at that index is undefined (i.e. $\perp$). Because the indices must be checked for these errors, `array` is strict in the bounds argument and in the indices of the association list, but nonstrict in the values. Thus, recurrences such as the following are possible:

```
a = array (1,100) ((1,1) : [(i, i * a!(i-1)) | i <- [2..100]])
```

Not every index within the bounds of the array need appear in the association list, but the values associated with indices that do not appear will be undefined. Figure 2 shows some examples that use the `array` constructor.

The `(!)` operator denotes array subscripting. The `bounds` function applied to an array returns its bounds. The functions `indices`, `elems`, and `assocs`, when applied to an array, return lists of the indices, elements, or associations, respectively, in index order. An array may be constructed from a pair of bounds and a list of values in index order using the function `listArray`.

If, in any dimension, the lower bound is greater than the upper bound, then the array is legal, but empty. Indexing an empty array always gives an array-bounds error, but `bounds` still yields the bounds with which the array was constructed.

6.1.1 Accumulated Arrays

Another array creation function, `accumArray`, relaxes the restriction that a given index may appear at most once in the association list, using an *accumulating function* which combines the values of associations with the same index. The first argument of `accumArray` is the accumulating function; the second is an initial value; the remaining two arguments are a bounds pair and an association list, as for the `array` function. For example, given a list of values of some index type, `hist` produces a histogram of the number of occurrences of each index within a specified range:

```
hist :: (Ix a, Num b) => (a,a) -> [a] -> Array a b
hist bnds is = accumArray (+) 0 bnds [(i, 1) | i<-is, inRange bnds i]
```

If the accumulating function is strict, then `accumArray` is strict in the values, as well as the indices, in the association list. Thus, unlike ordinary arrays, accumulated arrays should not in general be recursive.

6.2 Incremental Array Updates

The operator `(//)` takes an array and a list of pairs and returns an array identical to the left argument except that it has been updated by the associations in the right argument. (As with the `array` function, the indices in the association list must be unique for the updated elements to be defined.) For example, if `m` is a 1-origin, `n` by `n` matrix, then `m//[((i,i), 0) | i <- [1..n]]` is the same matrix, except with the diagonal zeroed.

`accum` f takes an array and an association list and accumulates pairs from the list into the array with the accumulating function f. Thus `accumArray` can be defined using `accum`:

```
accumArray f z b = accum f (array b [(i, z) | i <- range b])
```

6.3 Derived Arrays

The two functions `map` and `ixmap` derive new arrays from existing ones; they may be thought of as providing function composition on the left and right, respectively, with the mapping that the original array embodies. The `map` function transforms the array values while `ixmap` allows for transformations on array indices. Figure 3 shows some examples.

```
-- A rectangular subarray
subArray :: (Ix a) => (a,a) -> Array a b -> Array a b
subArray bnds = ixmap bnds (\i->i)

-- A row of a matrix
row :: (Ix a, Ix b) => a -> Array (a,b) c -> Array b c
row i x = ixmap (l',u') (\j->(i,j)) x where ((l,l'),(u,u')) = bounds x

-- Diagonal of a square matrix
diag :: (Ix a) => Array (a,a) b -> Array a b
diag x = ixmap (l,u) (\i->(i,i)) x
         where ((l,l'),(u,u')) | l == l' && u == u'  = bounds x

-- Projection of first components of an array of pairs
firstArray :: (Ix a) => Array a (b,c) -> Array a b
firstArray = map (\(x,y)->x)
```

Figure 3: Derived array examples

6.4 Library Array

```
module  Array (
    module Ix,  -- export all of Ix
    Array, array, listArray, (!), bounds, indices, elems, assocs,
    accumArray, (//), accum, ixmap ) where

import Ix
import List( (\\) )

infixl 9  !, //

data (Ix a) => Array a b = MkArray (a,a) (a -> b) deriving ()

array       :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
array b ivs =
    if and [inRange b i | (i,_) <- ivs]
        then MkArray b
                     (\j -> case [v | (i,v) <- ivs, i == j] of
                            [v]   -> v
                            []    -> error "Array.!: \
                                           \undefined array element"
                            _     -> error "Array.!: \
                                           \multiply defined array element")
        else error "Array.array: out-of-range array association"

listArray             :: (Ix a) => (a,a) -> [b] -> Array a b
listArray b vs        =  array b (zipWith (\ a b -> (a,b)) (range b) vs)
```

```
(!)                     :: (Ix a) => Array a b -> a -> b
(!) (MkArray _ f)       =  f

bounds                  :: (Ix a) => Array a b -> (a,a)
bounds (MkArray b _)    =  b

indices                 :: (Ix a) => Array a b -> [a]
indices                 =  range . bounds

elems                   :: (Ix a) => Array a b -> [b]
elems a                 =  [a!i | i <- indices a]

assocs                  :: (Ix a) => Array a b -> [(a,b)]
assocs a                =  [(i, a!i) | i <- indices a]

(//)                    :: (Ix a) => Array a b -> [(a,b)] -> Array a b
a // us                 =  array (bounds a)
                              ([(i,a!i) | i <- indices a \\ [i | (i,_) <- us]]
                               ++ us)

accum                   :: (Ix a) => (b -> c -> b) -> Array a b -> [(a,c)]
                                      -> Array a b
accum f                 =  foldl (\a (i,v) -> a // [(i,f (a!i) v)])

accumArray              :: (Ix a) => (b -> c -> b) -> b -> (a,a) -> [(a,c)]
                                      -> Array a b
accumArray f z b        =  accum f (array b [(i,z) | i <- range b])

ixmap                   :: (Ix a, Ix b) => (a,a) -> (a -> b) -> Array b c
                                            -> Array a c
ixmap b f a             = array b [(i, a ! f i) | i <- range b]

instance  (Ix a)          => Functor (Array a) where
    fmap fn (MkArray b f) =  MkArray b (fn . f)

instance  (Ix a, Eq b)  => Eq (Array a b)  where
    a == a'             =  assocs a == assocs a'

instance  (Ix a, Ord b) => Ord (Array a b)  where
    a <=  a'            =  assocs a <=  assocs a'

instance  (Ix a, Show a, Show b) => Show (Array a b)  where
    showsPrec p a = showParen (p > 9) (
                    showString "array " .
                    shows (bounds a) . showChar ' ' .
                    shows (assocs a)                  )
```

```
instance  (Ix a, Read a, Read b) => Read (Array a b)  where
    readsPrec p = readParen (p > 9)
           (\r -> [(array b as, u) | ("array",s) <- lex r,
                                     (b,t)       <- reads s,
                                     (as,u)      <- reads t   ])
```

7 List Utilities

```
module List (
    elemIndex, elemIndices,
    find, findIndex, findIndices,
    nub, nubBy, delete, deleteBy, (\\), deleteFirstsBy,
    union, unionBy, intersect, intersectBy,
    intersperse, transpose, partition, group, groupBy,
    inits, tails, isPrefixOf, isSuffixOf,
    mapAccumL, mapAccumR,
    sort, sortBy, insert, insertBy, maximumBy, minimumBy,
    genericLength, genericTake, genericDrop,
    genericSplitAt, genericIndex, genericReplicate,
    zip4, zip5, zip6, zip7,
    zipWith4, zipWith5, zipWith6, zipWith7,
    unzip4, unzip5, unzip6, unzip7, unfoldr,

    -- ...and what the Prelude exports
    []((:), []),
    map, (++), concat, filter,
    head, last, tail, init, null, length, (!!),
    foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1,
    iterate, repeat, replicate, cycle,
    take, drop, splitAt, takeWhile, dropWhile, span, break,
    lines, words, unlines, unwords, reverse, and, or,
    any, all, elem, notElem, lookup,
    sum, product, maximum, minimum, concatMap,
    zip, zip3, zipWith, zipWith3, unzip, unzip3
    ) where

infix 5 \\

elemIndex           :: Eq a => a -> [a] -> Maybe Int
elemIndices         :: Eq a => a -> [a] -> [Int]
find                :: (a -> Bool) -> [a] -> Maybe a
findIndex           :: (a -> Bool) -> [a] -> Maybe Int
findIndices         :: (a -> Bool) -> [a] -> [Int]
nub                 :: Eq a => [a] -> [a]
nubBy               :: (a -> a -> Bool) -> [a] -> [a]
delete              :: Eq a => a -> [a] -> [a]
deleteBy            :: (a -> a -> Bool) -> a -> [a] -> [a]
(\\)                :: Eq a => [a] -> [a] -> [a]
deleteFirstsBy      :: (a -> a -> Bool) -> [a] -> [a] -> [a]
union               :: Eq a => [a] -> [a] -> [a]
unionBy             :: (a -> a -> Bool) -> [a] -> [a] -> [a]
```

```
intersect           :: Eq a => [a] -> [a] -> [a]
intersectBy         :: (a -> a -> Bool) -> [a] -> [a] -> [a]
intersperse         :: a -> [a] -> [a]
transpose           :: [[a]] -> [[a]]
partition           :: (a -> Bool) -> [a] -> ([a],[a])
group               :: Eq a => [a] -> [[a]]
groupBy             :: (a -> a -> Bool) -> [a] -> [[a]]
inits               :: [a] -> [[a]]
tails               :: [a] -> [[a]]
isPrefixOf          :: Eq a => [a] -> [a] -> Bool
isSuffixOf          :: Eq a => [a] -> [a] -> Bool
mapAccumL           :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
mapAccumR           :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
unfoldr             :: (b -> Maybe (a,b)) -> b -> [a]
sort                :: Ord a => [a] -> [a]
sortBy              :: (a -> a -> Ordering) -> [a] -> [a]
insert              :: Ord a => a -> [a] -> [a]
insertBy            :: (a -> a -> Ordering) -> a -> [a] -> [a]
maximumBy           :: (a -> a -> Ordering) -> [a] -> a
minimumBy           :: (a -> a -> Ordering) -> [a] -> a
genericLength       :: Integral a => [b] -> a
genericTake         :: Integral a => a -> [b] -> [b]
genericDrop         :: Integral a => a -> [b] -> [b]
genericSplitAt      :: Integral a => a -> [b] -> ([b],[b])
genericIndex        :: Integral a => [b] -> a -> b
genericReplicate    :: Integral a => a -> b -> [b]

zip4                :: [a] -> [b] -> [c] -> [d] -> [(a,b,c,d)]
zip5                :: [a] -> [b] -> [c] -> [d] -> [e] -> [(a,b,c,d,e)]
zip6                :: [a] -> [b] -> [c] -> [d] -> [e] -> [f]
                         -> [(a,b,c,d,e,f)]
zip7                :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g]
                         -> [(a,b,c,d,e,f,g)]
zipWith4            :: (a->b->c->d->e) -> [a]->[b]->[c]->[d]->[e]
zipWith5            :: (a->b->c->d->e->f) ->
                       [a]->[b]->[c]->[d]->[e]->[f]
zipWith6            :: (a->b->c->d->e->f->g) ->
                       [a]->[b]->[c]->[d]->[e]->[f]->[g]
zipWith7            :: (a->b->c->d->e->f->g->h) ->
                       [a]->[b]->[c]->[d]->[e]->[f]->[g]->[h]
unzip4              :: [(a,b,c,d)] -> ([a],[b],[c],[d])
unzip5              :: [(a,b,c,d,e)] -> ([a],[b],[c],[d],[e])
unzip6              :: [(a,b,c,d,e,f)] -> ([a],[b],[c],[d],[e],[f])
unzip7              :: [(a,b,c,d,e,f,g)] -> ([a],[b],[c],[d],[e],[f],[g])
```

This library defines some lesser-used operations over lists.

7.1 Indexing lists

Function `elemIndex val list` returns the index of the first occurrence, if any, of `val` in `list` as `Just index`. `Nothing` is returned if `not (val ‘elem‘ list)`.

Function `elemIndices val list` returns an in-order list of indices, giving the occurrences of `val` in `list`.

Function `find` returns the first element of a list that satisfies a predicate, or Nothing, if there is no such element. `findIndex` returns the corresponding index. `findIndices` returns a list of all such indices.

7.2 “Set” operations

There are a number of “set” operations defined over the `List` type. `nub` (meaning “essence”) removes duplicates elements from a list. `delete`, `(\\)`, `union` and `intersect` preserve the invariant that lists don’t contain duplicates, provided that their first argument contains no duplicates.

- `delete x` removes the first occurrence of `x` from its list argument, e.g.,

  ```
  delete 'a' "banana" == "bnana"
  ```

- `(\\)` is list difference (non-associative). In the result of `xs \\ ys`, the first occurrence of each element of `ys` in turn (if any) has been removed from `xs`. Thus, `(xs ++ ys) \\ xs == ys`. `union` is list union, e.g.,

  ```
  "dog" `union` "cow" == "dogcw"
  ```

- `intersect` is list intersection, e.g.,

  ```
  intersect [1,2,3,4] `intersect` [2,4,6,8] == [2,4]
  ```

7.3 List transformations

- `intersperse sep` inserts `sep` between the elements of its list argument, e.g.,

  ```
  intersperse ',' "abcde" == "a,b,c,d,e"
  ```

- `transpose` transposes the rows and columns of its argument, e.g.,

```
transpose [[1,2,3],[4,5,6]] == [[1,4],[2,5],[3,6]]
```

- `partition` takes a predicate and a list and returns a pair of lists: those elements of the argument list that do and do not satisfy the predicate, respectively; i.e.,

```
partition p xs == (filter p xs, filter (not . p) xs)
```

- `sort`/`sortBy` implement a stable sorting algorithm, here specified in terms of the `insertBy` function, which inserts objects into a list according to the specified ordering relation.

- `group` splits its list argument into a list of lists of equal, adjacent elements. For exmaple

```
group "Mississippi" == ["M","i","ss","i","ss","i","pp","i"]
```

- `inits` returns the list of initial segments of its argument list, shortest first.

```
inits "abc" == ["","a","ab","abc"]
```

- `tails` returns the list of all final segments of its argument list, longest first.

```
tails "abc" == ["abc", "bc", "c",""]
```

- `mapAccumL f s l` applies `f` to an accumulating "state" parameter `s` and to each element of `l` in turn.

- `mapAccumR` is similar to `mapAccumL` except that the list is processed from right-to-left rather than left-to-right.

7.4 `unfoldr`

The `unfoldr` function undoes a `foldr` operation. Note that, in general, only invertible functions can be unfolded.

```
unfoldr f' (foldr f z xs) == xs
```

if the following holds:

```
f' (f x y) = Just (x,y)
f' z       = Nothing
```

7.5 Predicates

`isPrefixOf` and `isSuffixOf` check whether the first argument is a prefix (resp. suffix) of the second argument.

7.6 The "By" operations

By convention, overloaded functions have a non-overloaded counterpart whose name is suffixed with "By". For example, the function `nub` could be defined as follows:

```
nub                     :: (Eq a) => [a] -> [a]
nub []                  =  []
nub (x:xs)              =  x : nub (filter (\y -> x /= y) xs)
```

However, the equality method may not be appropriate in all situations. The function:

```
nubBy                   :: (a -> a -> Bool) -> [a] -> [a]
nubBy eq []             =  []
nubBy eq (x:xs)         =  x : nubBy eq (filter (\y -> not (eq x y)) xs)
```

allows the programmer to supply their own equality test. When the "By" function replaces an `Eq` context by a binary predicate, the predicate is assumed to define an equivalence; when the "By" function replaces an `Ord` context by a binary predicate, the predicate is assumed to define a total ordering.

The "By" variants are as follows: `nubBy`, `deleteBy`, `unionBy`, `intersectBy`, `groupBy`, `sortBy`, `insertBy`, `maximumBy`, `minimumBy`. The library does not provide `elemBy`, because `any (eq x)` does the same job as `elemBy eq x` would. A handful of overloaded functions (`elemIndex`, `elemIndices`, `isPrefixOf`, `isSuffixOf`) were not considered important enough to have "By" variants.

7.7 The "generic" operations

The prefix "`generic`" indicates an overloaded function that is a generalised version of a `Prelude` function. For example,

```
genericLength       :: Integral a => [b] -> a
```

is a generalised verion of `length`.

The "`generic`" operations are as follows: `genericLength`, `genericTake`, `genericDrop`, `genericSplitAt`, `genericIndex`, `genericReplicate`.

7.8 Library List

```
module List (
    elemIndex, elemIndices,
    find, findIndex, findIndices,
    nub, nubBy, delete, deleteBy, (\\),
    union, unionBy, intersect, intersectBy,
    intersperse, transpose, partition, group, groupBy,
    inits, tails, isPrefixOf, isSuffixOf,
    mapAccumL, mapAccumR,
    sort, sortBy, insert, insertBy, maximumBy, minimumBy,
    genericLength, genericTake, genericDrop,
    genericSplitAt, genericIndex, genericReplicate,
    zip4, zip5, zip6, zip7,
    zipWith4, zipWith5, zipWith6, zipWith7,
    unzip4, unzip5, unzip6, unzip7, unfoldr,

    -- ...and what the Prelude exports
    []((:), []),
    map, (++), concat, filter,
    head, last, tail, init, null, length, (!!),
    foldl, foldl1, scanl, scanl1, foldr, foldr1, scanr, scanr1,
    iterate, repeat, replicate, cycle,
    take, drop, splitAt, takeWhile, dropWhile, span, break,
    lines, words, unlines, unwords, reverse, and, or,
    any, all, elem, notElem, lookup,
    sum, product, maximum, minimum, concatMap,
    zip, zip3, zipWith, zipWith3, unzip, unzip3
    ) where

import Maybe( listToMaybe )

infix 5 \\

elemIndex               :: Eq a => a -> [a] -> Maybe Int
elemIndex x             =  findIndex (x ==)

elemIndices             :: Eq a => a -> [a] -> [Int]
elemIndices x           =  findIndices (x ==)

find                    :: (a -> Bool) -> [a] -> Maybe a
find p                  =  listToMaybe . filter p

findIndex               :: (a -> Bool) -> [a] -> Maybe Int
findIndex p             =  listToMaybe . findIndices p

findIndices             :: (a -> Bool) -> [a] -> [Int]
findIndices p xs        =  [ i | (x,i) <- zip xs [0..], p x ]
```

```
nub                     :: Eq a => [a] -> [a]
nub                     =  nubBy (==)

nubBy                   :: (a -> a -> Bool) -> [a] -> [a]
nubBy eq []             =  []
nubBy eq (x:xs)         =  x : nubBy eq (filter (\y -> not (eq x y)) xs)

delete                  :: Eq a => a -> [a] -> [a]
delete                  =  deleteBy (==)

deleteBy                :: (a -> a -> Bool) -> a -> [a] -> [a]
deleteBy eq x []        = []
deleteBy eq x (y:ys)    = if x `eq` y then ys else y : deleteBy eq x ys

(\\)                    :: Eq a => [a] -> [a] -> [a]
(\\)                    =  foldl (flip delete)

deleteFirstsBy          :: (a -> a -> Bool) -> [a] -> [a] -> [a]
deleteFirstsBy eq       =  foldl (flip (deleteBy eq))

union                   :: Eq a => [a] -> [a] -> [a]
union                   =  unionBy (==)

unionBy                 :: (a -> a -> Bool) -> [a] -> [a] -> [a]
unionBy eq xs ys        =  xs ++ foldl (flip (deleteBy eq)) (nubBy eq ys) xs

intersect               :: Eq a => [a] -> [a] -> [a]
intersect               =  intersectBy (==)

intersectBy             :: (a -> a -> Bool) -> [a] -> [a] -> [a]
intersectBy eq xs ys    =  [x | x <- xs, any (eq x) ys]

intersperse             :: a -> [a] -> [a]
intersperse sep []      =  []
intersperse sep [x]     =  [x]
intersperse sep (x:xs)  =  x : sep : intersperse sep xs

-- transpose is lazy in both rows and columns,
--       and works for non-rectangular 'matrices'
-- For example, transpose [[1,2],[3,4,5],[]]  =  [[1,3],[2,4],[5]]
-- Note that [h | (h:t) <- xss] is not the same as (map head xss)
--      because the former discards empty sublists inside xss
transpose                :: [[a]] -> [[a]]
transpose []             = []
transpose ([]     : xss) = transpose xss
transpose ((x:xs) : xss) = (x : [h | (h:t) <- xss]) :
                           transpose (xs : [t | (h:t) <- xss])
```

```
partition               :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs          =  foldr select ([],[]) xs
                           where select x (ts,fs) | p x       = (x:ts,fs)
                                                  | otherwise = (ts, x:fs)

-- group splits its list argument into a list of lists of equal, adjacent
-- elements.  e.g.,
-- group "Mississippi" == ["M","i","ss","i","ss","i","pp","i"]
group                   :: Eq a => [a] -> [[a]]
group                   =  groupBy (==)

groupBy                 :: (a -> a -> Bool) -> [a] -> [[a]]
groupBy eq []           =  []
groupBy eq (x:xs)       =  (x:ys) : groupBy eq zs
                           where (ys,zs) = span (eq x) xs

-- inits xs returns the list of initial segments of xs, shortest first.
-- e.g., inits "abc" == ["","a","ab","abc"]
inits                   :: [a] -> [[a]]
inits []                =  [[]]
inits (x:xs)            =  [[]] ++ map (x:) (inits xs)

-- tails xs returns the list of all final segments of xs, longest first.
-- e.g., tails "abc" == ["abc", "bc", "c",""]
tails                   :: [a] -> [[a]]
tails []                =  [[]]
tails xxs@(_:xs)        =  xxs : tails xs

isPrefixOf               :: Eq a => [a] -> [a] -> Bool
isPrefixOf []     _      =  True
isPrefixOf _      []     =  False
isPrefixOf (x:xs) (y:ys) =  x == y && isPrefixOf xs ys

isSuffixOf              :: Eq a => [a] -> [a] -> Bool
isSuffixOf x y          =  reverse x `isPrefixOf` reverse y

mapAccumL               :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
mapAccumL f s []        =  (s, [])
mapAccumL f s (x:xs)    =  (s'',y:ys)
                           where (s', y ) = f s x
                                 (s'',ys) = mapAccumL f s' xs

mapAccumR               :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
mapAccumR f s []        =  (s, [])
mapAccumR f s (x:xs)    =  (s'', y:ys)
                           where (s'',y ) = f s' x
                                 (s', ys) = mapAccumR f s xs
```

```
unfoldr                 :: (b -> Maybe (a,b)) -> b -> [a]
unfoldr f b             = case f b of
                                Nothing    -> []
                                Just (a,b) -> a : unfoldr f b

sort                    :: (Ord a) => [a] -> [a]
sort                    =  sortBy compare

sortBy                  :: (a -> a -> Ordering) -> [a] -> [a]
sortBy cmp              =  foldr (insertBy cmp) []

insert                  :: (Ord a) => a -> [a] -> [a]
insert                  = insertBy compare

insertBy                :: (a -> a -> Ordering) -> a -> [a] -> [a]
insertBy cmp x []       =  [x]
insertBy cmp x ys@(y:ys')
                        =  case cmp x y of
                                GT -> y : insertBy cmp x ys'
                                _  -> x : ys

maximumBy               :: (a -> a -> a) -> [a] -> a
maximumBy max []        =  error "List.maximumBy: empty list"
maximumBy max xs        =  foldl1 max xs

minimumBy               :: (a -> a -> a) -> [a] -> a
minimumBy min []        =  error "List.minimumBy: empty list"
minimumBy min xs        =  foldl1 min xs

genericLength           :: (Integral a) => [b] -> a
genericLength []        =  0
genericLength (x:xs)    =  1 + genericLength xs

genericTake             :: (Integral a) => a -> [b] -> [b]
genericTake _ []        =  []
genericTake 0 _         =  []
genericTake n (x:xs)
   | n > 0              =  x : genericTake (n-1) xs
   | otherwise          =  error "List.genericTake: negative argument"

genericDrop             :: (Integral a) => a -> [b] -> [b]
genericDrop 0 xs        =  xs
genericDrop _ []        =  []
genericDrop n (_:xs)
   | n > 0              =  genericDrop (n-1) xs
   | otherwise          =  error "List.genericDrop: negative argument"
```

```
genericSplitAt          :: (Integral a) => a -> [b] -> ([b],[b])
genericSplitAt 0 xs     =  ([],xs)
genericSplitAt _ []     =  ([],[])
genericSplitAt n (x:xs)
   | n > 0              =  (x:xs',xs'')
   | otherwise          =  error "List.genericSplitAt: negative argument"
       where (xs',xs'') =  genericSplitAt (n-1) xs

genericIndex            :: (Integral a) => [b] -> a -> b
genericIndex (x:_)  0   =  x
genericIndex (_:xs) n
        | n > 0         =  genericIndex xs (n-1)
        | otherwise     =  error "List.genericIndex: negative argument"
genericIndex _ _        =  error "List.genericIndex: index too large"

genericReplicate        :: (Integral a) => a -> b -> [b]
genericReplicate n x    =  genericTake n (repeat x)

zip4                    :: [a] -> [b] -> [c] -> [d] -> [(a,b,c,d)]
zip4                    =  zipWith4 (,,,)

zip5                    :: [a] -> [b] -> [c] -> [d] -> [e] -> [(a,b,c,d,e)]
zip5                    =  zipWith5 (,,,,)

zip6                    :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] ->
                              [(a,b,c,d,e,f)]
zip6                    =  zipWith6 (,,,,,)

zip7                    :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] ->
                              [g] -> [(a,b,c,d,e,f,g)]
zip7                    =  zipWith7 (,,,,,,)

zipWith4                :: (a->b->c->d->e) -> [a]->[b]->[c]->[d]->[e]
zipWith4 z (a:as) (b:bs) (c:cs) (d:ds)
                        =  z a b c d : zipWith4 z as bs cs ds
zipWith4 _ _ _ _ _      =  []

zipWith5                :: (a->b->c->d->e->f) ->
                           [a]->[b]->[c]->[d]->[e]->[f]
zipWith5 z (a:as) (b:bs) (c:cs) (d:ds) (e:es)
                        =  z a b c d e : zipWith5 z as bs cs ds es
zipWith5 _ _ _ _ _ _    =  []

zipWith6                :: (a->b->c->d->e->f->g) ->
                           [a]->[b]->[c]->[d]->[e]->[f]->[g]
zipWith6 z (a:as) (b:bs) (c:cs) (d:ds) (e:es) (f:fs)
                        =  z a b c d e f : zipWith6 z as bs cs ds es fs
zipWith6 _ _ _ _ _ _ _  =  []
```

```
zipWith7                 :: (a->b->c->d->e->f->g->h) ->
                            [a]->[b]->[c]->[d]->[e]->[f]->[g]->[h]
zipWith7 z (a:as) (b:bs) (c:cs) (d:ds) (e:es) (f:fs) (g:gs)
                   =  z a b c d e f g : zipWith7 z as bs cs ds es fs gs
zipWith7 _ _ _ _ _ _ _ _ = []

unzip4                   :: [(a,b,c,d)] -> ([a],[b],[c],[d])
unzip4                   =  foldr (\(a,b,c,d) ~(as,bs,cs,ds) ->
                                        (a:as,b:bs,c:cs,d:ds))
                                 ([],[],[],[])

unzip5                   :: [(a,b,c,d,e)] -> ([a],[b],[c],[d],[e])
unzip5                   =  foldr (\(a,b,c,d,e) ~(as,bs,cs,ds,es) ->
                                        (a:as,b:bs,c:cs,d:ds,e:es))
                                 ([],[],[],[],[])

unzip6                   :: [(a,b,c,d,e,f)] -> ([a],[b],[c],[d],[e],[f])
unzip6                   =  foldr (\(a,b,c,d,e,f) ~(as,bs,cs,ds,es,fs) ->
                                        (a:as,b:bs,c:cs,d:ds,e:es,f:fs))
                                 ([],[],[],[],[],[])

unzip7           :: [(a,b,c,d,e,f,g)] -> ([a],[b],[c],[d],[e],[f],[g])
unzip7           =  foldr (\(a,b,c,d,e,f,g) ~(as,bs,cs,ds,es,fs,gs) ->
                                (a:as,b:bs,c:cs,d:ds,e:es,f:fs,g:gs))
                         ([],[],[],[],[],[],[])
```

8 Maybe Utilities

```
module Maybe(
    isJust, isNothing,
    fromJust, fromMaybe, listToMaybe, maybeToList,
    catMaybes, mapMaybe,

    -- ...and what the Prelude exports
    Maybe(Nothing, Just),
    maybe
  ) where

isJust, isNothing    :: Maybe a -> Bool
fromJust             :: Maybe a -> a
fromMaybe            :: a -> Maybe a -> a
listToMaybe          :: [a] -> Maybe a
maybeToList          :: Maybe a -> [a]
catMaybes            :: [Maybe a] -> [a]
mapMaybe             :: (a -> Maybe b) -> [a] -> [b]
```

The type constructor `Maybe` is defined in `Prelude` as

```
data Maybe a = Nothing | Just a
```

The purpose of the `Maybe` type is to provide a method of dealing with illegal or optional values without terminating the program, as would happen if `error` were used, and without using `IOError` from the `IO` monad, which would cause the expression to become monadic. A correct result is encapsulated by wrapping it in `Just`; an incorrect result is returned as `Nothing`.

Other operations on `Maybe` are provided as part of the monadic classes in the Prelude.

8.1 Library Maybe

```
module Maybe(
    isJust, isNothing,
    fromJust, fromMaybe, listToMaybe, maybeToList,
    catMaybes, mapMaybe,

    -- ...and what the Prelude exports
    Maybe(Nothing, Just),
    maybe
  ) where

isJust                 :: Maybe a -> Bool
isJust (Just a)        =  True
isJust Nothing         =  False

isNothing              :: Maybe a -> Bool
isNothing              =  not . isJust

fromJust               :: Maybe a -> a
fromJust (Just a)      =  a
fromJust Nothing       =  error "Maybe.fromJust: Nothing"

fromMaybe              :: a -> Maybe a -> a
fromMaybe d Nothing    =  d
fromMaybe d (Just a)   =  a

maybeToList            :: Maybe a -> [a]
maybeToList Nothing    =  []
maybeToList (Just a)   =  [a]

listToMaybe            :: [a] -> Maybe a
listToMaybe []         =  Nothing
listToMaybe (a:_)      =  Just a

catMaybes              :: [Maybe a] -> [a]
catMaybes ms           =  [ m | Just m <- ms ]

mapMaybe               :: (a -> Maybe b) -> [a] -> [b]
mapMaybe f             =  catMaybes . map f
```

9 Character Utilities

```
module Char (
    isAscii, isLatin1, isControl, isPrint, isSpace, isUpper, isLower,
    isAlpha, isDigit, isOctDigit, isHexDigit, isAlphaNum,
    digitToInt, intToDigit,
    toUpper, toLower,
    ord, chr,
    readLitChar, showLitChar, lexLitChar

        -- ...and what the Prelude exports
    Char, String
    ) where

isAscii, isLatin1, isControl, isPrint, isSpace, isUpper, isLower,
 isAlpha, isDigit, isOctDigit, isHexDigit, isAlphaNum :: Char -> Bool

toUpper, toLower        :: Char -> Char

digitToInt :: Char -> Int
intToDigit :: Int -> Char

ord        :: Char -> Int
chr        :: Int  -> Char

lexLitChar  :: ReadS String
readLitChar :: ReadS Char
showLitChar :: Char -> ShowS
```

This library provides a limited set of operations on the Unicode character set. The first 128 entries of this character set are identical to the ASCII set; with the next 128 entries comes the remainder of the Latin-1 character set. This module offers only a limited view of the full Unicode character set; the full set of Unicode character attributes is not accessible in this library.

Unicode characters may be divided into five general categories: non-printing, lower case alphabetic, other alphabetic, numeric digits, and other printable characters. For the purposes of Haskell, any alphabetic character which is not lower case is treated as upper case (Unicode actually has three cases: upper, lower, and title). Numeric digits may be part of identifiers but digits outside the ASCII range are not used by the reader to represent numbers.

For each sort of Unicode character, here are the predicates which return `True`:

Character Type	Predicates			
Lower Case Alphabetic	`isPrint`	`isAlphaNum`	`isAlpha`	`isLower`
Other Alphabetic	`isPrint`	`isAlphaNum`	`isAlpha`	`isUpper`
Digits	`isPrint`	`isAlphaNum`		
Other Printable	`isPrint`			
Non-printing				

The `isDigit`, `isOctDigit`, and `isHexDigit` functions select only ASCII characters. `intToDigit` and `digitToInt` convert between a single digit `Char` and the corresponding `Int`. `digitToInt` operates fails unless its argument satisfies `isHexDigit`, but recognises both upper and lower-case hexadecimal digits (i.e. `'0'..'9'`, `'a'..'f'`, `'A'..'F'`). `intToDigit` fails unless its argument is in the range `0..15`, and generates lower-case hexadecmial digits.

The `isSpace` function recognizes only white characters in the Latin-1 range.

The function `showLitChar` converts a character to a string using only printable characters, using Haskell source-language escape conventions. The function `readLitChar` does the reverse.

Function `toUpper` converts a letter to the corresponding upper-case letter, leaving any other character unchanged. Any Unicode letter which has an upper-case equivalent is transformed. Similarly, `toLower` converts a letter to the corresponding lower-case letter, leaving any other character unchanged.

The `ord` and `chr` functions are `fromEnum` and `toEnum` restricted to the type `Char`.

9.1 Library Char

```
module Char (
    isAscii, isLatin1, isControl, isPrint, isSpace, isUpper, isLower,
    isAlpha, isDigit, isOctDigit, isHexDigit, isAlphaNum,
    digitToInt, intToDigit,
    toUpper, toLower,
    ord, chr,
    readLitChar, showLitChar, lexLitChar,

        -- ...and what the Prelude exports
    Char, String
    ) where

import Array  -- used for character name table.
import Numeric (readDec, readOct, lexDigits, readHex)
import UnicodePrims  -- source of primitive Unicode functions.

-- Character-testing operations
isAscii, isControl, isPrint, isSpace, isUpper, isLower,
 isAlpha, isDigit, isOctDigit, isHexDigit, isAlphaNum :: Char -> Bool

isAscii c                =  c < '\x80'

isLatin1 c               =  c <= '\xff'

isControl c              =  c < ' ' || c >= '\DEL' && c <= '\x9f'

isPrint                  =  primUnicodeIsPrint

isSpace c                =  c `elem` " \t\n\r\f\v\xA0"
        -- Only Latin-1 spaces recognized

isUpper                  =  primUnicodeIsUpper  -- 'A'..'Z'

isLower                  =  primUnicodeIsLower  -- 'a'..'z'

isAlpha c                =  isUpper c || isLower c

isDigit c                =  c >= '0' && c <= '9'

isOctDigit c             =  c >= '0' && c <= '7'

isHexDigit c             =  isDigit c || c >= 'A' && c <= 'F' ||
                                         c >= 'a' && c <= 'f'

isAlphaNum               =  primUnicodeIsAlphaNum
```

```
-- Digit conversion operations
digitToInt :: Char -> Int
digitToInt c
  | isDigit c            =  fromEnum c - fromEnum '0'
  | c >= 'a' && c <= 'f' =  fromEnum c - fromEnum 'a' + 10
  | c >= 'A' && c <= 'F' =  fromEnum c - fromEnum 'A' + 10
  | otherwise            =  error "Char.digitToInt: not a digit"

intToDigit :: Int -> Char
intToDigit i
  | i >= 0  && i <=  9   =  toEnum (fromEnum '0' + i)
  | i >= 10 && i <= 15   =  toEnum (fromEnum 'a' + i - 10)
  | otherwise            =  error "Char.intToDigit: not a digit"

-- Case-changing operations
toUpper                  :: Char -> Char
toUpper                  =  primUnicodeToUpper

toLower                  :: Char -> Char
toLower                  =  primUnicodeToLower

-- Character code functions
ord                     :: Char -> Int
ord                     =  fromEnum

chr                     :: Int  -> Char
chr                     =  toEnum
```

```
-- Text functions
readLitChar             :: ReadS Char
readLitChar ('\\':s)    =  readEsc s
        where
        readEsc ('a':s)  = [('\a',s)]
        readEsc ('b':s)  = [('\b',s)]
        readEsc ('f':s)  = [('\f',s)]
        readEsc ('n':s)  = [('\n',s)]
        readEsc ('r':s)  = [('\r',s)]
        readEsc ('t':s)  = [('\t',s)]
        readEsc ('v':s)  = [('\v',s)]
        readEsc ('\\':s) = [('\\',s)]
        readEsc ('"':s)  = [('"',s)]
        readEsc ('\'':s) = [('\'',s)]
        readEsc ('^':c:s) | c >= '@' && c <= '_'
                         = [(chr (ord c - ord '@'), s)]
        readEsc s@(d:_) | isDigit d
                         = [(chr n, t) | (n,t) <- readDec s]
        readEsc ('o':s)  = [(chr n, t) | (n,t) <- readOct s]
        readEsc ('x':s)  = [(chr n, t) | (n,t) <- readHex s]
        readEsc s@(c:_) | isUpper c
                         = let table = ('\DEL', "DEL") : assocs asciiTab
                           in case [(c,s') | (c, mne) <- table,
                                             ([],s') <- [match mne s]]
                              of (pr:_) -> [pr]
                                 []     -> []
        readEsc _        = []

        match                         :: (Eq a) => [a] -> [a] -> ([a],[a])
        match (x:xs) (y:ys) | x == y  =  match xs ys
        match xs     ys               =  (xs,ys)

readLitChar (c:s)       =  [(c,s)]
```

```
showLitChar               :: Char -> ShowS
showLitChar c | c > '\DEL' =  showChar '\\' .
                              protectEsc isDigit (shows (ord c))
showLitChar '\DEL'         =  showString "\\DEL"
showLitChar '\\'           =  showString "\\\\"
showLitChar c | c >= ' '   =  showChar c
showLitChar '\a'           =  showString "\\a"
showLitChar '\b'           =  showString "\\b"
showLitChar '\f'           =  showString "\\f"
showLitChar '\n'           =  showString "\\n"
showLitChar '\r'           =  showString "\\r"
showLitChar '\t'           =  showString "\\t"
showLitChar '\v'           =  showString "\\v"
showLitChar '\SO'          =  protectEsc (== 'H') (showString "\\SO")
showLitChar c              =  showString ('\\' : asciiTab!c)

protectEsc p f             = f . cont
                             where cont s@(c:_) | p c = "\\&" ++ s
                                   cont s             = s
asciiTab = listArray ('\NUL', ' ')
           ["NUL", "SOH", "STX", "ETX", "EOT", "ENQ", "ACK", "BEL",
            "BS",  "HT",  "LF",  "VT",  "FF",  "CR",  "SO",  "SI",
            "DLE", "DC1", "DC2", "DC3", "DC4", "NAK", "SYN", "ETB",
            "CAN", "EM",  "SUB", "ESC", "FS",  "GS",  "RS",  "US",
            "SP"]

lexLitChar          :: ReadS String
lexLitChar ('\\':s) =  [('\\':esc, t) | (esc,t) <- lexEsc s]
        where
          lexEsc (c:s)     | c `elem` "abfnrtv\\\"'" = [([c],s)]
          lexEsc s@(d:_)   | isDigit d               = lexDigits s
          lexEsc ('^':c:s) | c >= '@' && c <= '_'    = [(['^',c],s)]
          -- Very crude approximation to \XYZ.  Let readers work this out.
          lexEsc s@(c:_)   | isUpper c               = [span isCharName s]
          lexEsc _                                   = []
          isCharName c = isUpper c || isDigit c

lexLitChar (c:s)    =  [([c],s)]
lexLitChar ""       =  []
```

10 Monad Utilities

```
module Monad (
    MonadPlus(mzero, mplus),
    join, guard, when, unless, ap,
    msum,
    filterM, mapAndUnzipM, zipWithM, zipWithM_, foldM,
    liftM, liftM2, liftM3, liftM4, liftM5,

    -- ...and what the Prelude exports
    Monad((>>=), (>>), return, fail),
    Functor(fmap),
    mapM, mapM_, sequence, sequence_, (=<<),
    ) where

class  Monad m => MonadPlus m  where
    mzero  :: m a
    mplus  :: m a -> m a -> m a

join             :: Monad m => m (m a) -> m a
guard            :: MonadPlus m => Bool -> m ()
when             :: Monad m => Bool -> m () -> m ()
unless           :: Monad m => Bool -> m () -> m ()
ap               :: Monad m => m (a -> b) -> m a -> m b

mapAndUnzipM     :: Monad m => (a -> m (b,c)) -> [a] -> m ([b], [c])
zipWithM         :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM_        :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m ()
foldM            :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
filterM          :: Monad m => (a -> m Bool) -> [a] -> m [a]

msum             :: MonadPlus m => [m a] -> m a

liftM            :: Monad m => (a -> b) -> (m a -> m b)
liftM2           :: Monad m => (a -> b -> c) -> (m a -> m b -> m c)
liftM3           :: Monad m => (a -> b -> c -> d) ->
                               (m a -> m b -> m c -> m d)
liftM4           :: Monad m => (a -> b -> c -> d -> e) ->
                               (m a -> m b -> m c -> m d -> m e)
liftM5           :: Monad m => (a -> b -> c -> d -> e -> f) ->
                               (m a -> m b -> m c -> m d -> m e -> m f)
```

The `Monad` library defines the `MonadPlus` class, and provides some useful operations on monads.

10.1 Naming conventions

The functions in this library use the following naming conventions:

- A postfix "`M`" always stands for a function in the Kleisli category: `m` is added to function results (modulo currying) and nowhere else. So, for example,

```
filter  ::               (a ->   Bool) -> [a] ->   [a]
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

- A postfix "`_`" changes the result type from `(m a)` to `(m ())`. Thus (in the `Prelude`):

```
sequence  :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
```

- A prefix "`m`" generalises an existing function to a monadic form. Thus, for example:

```
sum  :: Num a       => [a]   -> a
msum :: MonadPlus m => [m a] -> m a
```

10.2 Class `MonadPlus`

The `MonadPlus` class is defined as follows:

```
class  (Monad m) => MonadPlus m  where
    mzero  :: m a
    mplus  :: m a -> m a -> m a
```

The class methods `mzero` and `mplus` are the zero and plus of the monad.

Lists and the `Maybe` type are instances of `MonadPlus`, thus:

```
instance  MonadPlus Maybe  where
    mzero                 = Nothing
    Nothing `mplus` ys    = ys
    xs      `mplus` ys    = xs

instance  MonadPlus []  where
    mzero = []
    mplus = (++)
```

10.3 Functions

The `join` function is the conventional monad join operator. It is used to remove one level of monadic structure, projecting its bound argument into the outer level.

The `mapAndUnzipM` function maps its first argument over a list, returning the result as a pair of lists. This function is mainly used with complicated data structures or a state-transforming monad.

The `zipWithM` function generalises `zipWith` to arbitrary monads. For instance the following function displays a file, prefixing each line with its line number,

```
listFile :: String -> IO ()
listFile nm =
  do cts <- openFile nm
     zipWithM_ (\i line -> do putStr (show i); putStr ": "; putStrLn line)
               [1..]
               (lines cts)
```

The `foldM` function is analogous to `foldl`, except that its result is encapsulated in a monad. Note that `foldM` works from left-to-right over the list arguments. This could be an issue where `(>>)` and the "folded function" are not commutative.

```
    foldM f a1 [x1, x2, ..., xm ]
==
    do
      a2 <- f a1 x1
      a3 <- f a2 x2
      ...
      f am xm
```

If right-to-left evaluation is required, the input list should be reversed.

The `when` and `unless` functions provide conditional execution of monadic expressions. For example,

```
when debug (putStr "Debugging\n")
```

will output the string `"Debugging\n"` if the Boolean value `debug` is `True`, and otherwise do nothing.

The monadic lifting operators promote a function to a monad. The function arguments are scanned left to right. For example,

```
liftM2 (+) [0,1] [0,2] = [0,2,1,3]
liftM2 (+) (Just 1) Nothing = Nothing
```

In many situations, the `liftM` operations can be replaced by uses of `ap`, which promotes function application.

```
return f `ap` x1 `ap` ... `ap` xn
```

is equivalent to

```
liftMn f x1 x2 ... xn
```

10.4 Library `Monad`

```
module Monad (
    MonadPlus(mzero, mplus),
    join, guard, when, unless, ap,
    msum,
    filterM, mapAndUnzipM, zipWithM, zipWithM_, foldM,
    liftM, liftM2, liftM3, liftM4, liftM5,

    -- ...and what the Prelude exports
    Monad((>>=), (>>), return, fail),
    Functor(fmap),
    mapM, mapM_, sequence, sequence_, (=<<),
    ) where

-- The MonadPlus class definition

class  (Monad m) => MonadPlus m  where
    mzero  :: m a
    mplus  :: m a -> m a -> m a

-- Instances of MonadPlus

instance  MonadPlus Maybe  where
    mzero                 = Nothing

    Nothing `mplus` ys    =  ys
    xs      `mplus` ys    =  xs

instance  MonadPlus []  where
    mzero =  []
    mplus = (++)

-- Functions


msum             :: MonadPlus m => [m a] -> m a
msum xs          =  foldr mplus mzero xs

join             :: (Monad m) => m (m a) -> m a
join x           =  x >>= id

when             :: (Monad m) => Bool -> m () -> m ()
when p s         =  if p then s else return ()

unless           :: (Monad m) => Bool -> m () -> m ()
unless p s       =  when (not p) s

ap               :: (Monad m) => m (a -> b) -> m a -> m b
ap               =  liftM2 ($)
```

```
guard             :: MonadPlus m => Bool -> m ()
guard p           =  if p then return () else mzero

mapAndUnzipM     :: (Monad m) => (a -> m (b,c)) -> [a] -> m ([b], [c])
mapAndUnzipM f xs = sequence (map f xs) >>= return . unzip

zipWithM          :: (Monad m) => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM f xs ys =  sequence (zipWith f xs ys)

zipWithM_          :: (Monad m) => (a -> b -> m c) -> [a] -> [b] -> m ()
zipWithM_ f xs ys =  sequence_ (zipWith f xs ys)

foldM             :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
foldM f a []      =  return a
foldM f a (x:xs) =  f a x >>= \ y -> foldM f y xs

filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p []      = return []
filterM p (x:xs) = do { b  <- p x;
                        ys <- filterM p xs;
                        return (if b then (x:ys) else ys)
                   }

liftM             :: (Monad m) => (a -> b) -> (m a -> m b)
liftM f           =  \a -> do { a' <- a; return (f a') }

liftM2            :: (Monad m) => (a -> b -> c) -> (m a -> m b -> m c)
liftM2 f          =  \a b -> do { a' <- a; b' <- b; return (f a' b') }

liftM3            :: (Monad m) => (a -> b -> c -> d) ->
                                  (m a -> m b -> m c -> m d)
liftM3 f          =  \a b c -> do { a' <- a; b' <- b; c' <- c;
                                    return (f a' b' c') }

liftM4            :: (Monad m) => (a -> b -> c -> d -> e) ->
                                  (m a -> m b -> m c -> m d -> m e)
liftM4 f          =  \a b c d -> do { a' <- a; b' <- b; c' <- c; d' <- d;
                                      return (f a' b' c' d') }

liftM5            :: (Monad m) => (a -> b -> c -> d -> e -> f) ->
                                  (m a -> m b -> m c -> m d -> m e -> m f)
liftM5 f          =  \a b c d e -> do { a' <- a; b' <- b; c' <- c; d' <- d;
                                        e' <- e; return (f a' b' c' d' e') }
```

11 Input/Output

```
module IO (
    Handle, HandlePosn,
    IOMode(ReadMode,WriteMode,AppendMode,ReadWriteMode),
    BufferMode(NoBuffering,LineBuffering,BlockBuffering),
    SeekMode(AbsoluteSeek,RelativeSeek,SeekFromEnd),
    stdin, stdout, stderr,
    openFile, hClose, hFileSize, hIsEOF, isEOF,
    hSetBuffering, hGetBuffering, hFlush,
    hGetPosn, hSetPosn, hSeek,
    hWaitForInput, hReady, hGetChar, hGetLine, hLookAhead, hGetContents,
    hPutChar, hPutStr, hPutStrLn, hPrint,
    hIsOpen, hIsClosed, hIsReadable, hIsWritable, hIsSeekable,
    isAlreadyExistsError, isDoesNotExistError, isAlreadyInUseError,
    isFullError, isEOFError,
    isIllegalOperation, isPermissionError, isUserError,
    ioeGetErrorString, ioeGetHandle, ioeGetFileName,
    try, bracket, bracket_

    -- ...and what the Prelude exports
    IO, FilePath, IOError, ioError, userError, catch, interact,
    putChar, putStr, putStrLn, print, getChar, getLine, getContents,
    readFile, writeFile, appendFile, readIO, readLn
    ) where

import Ix(Ix)

data Handle = ...                      -- implementation-dependent
instance Eq Handle where ...
instance Show Handle where ..          -- implementation-dependent

data HandlePosn = ...                  -- implementation-dependent
instance Eq HandlePosn where ...
instance Show HandlePosn where ---     -- implementation-dependent

data IOMode      =  ReadMode | WriteMode | AppendMode | ReadWriteMode
                    deriving (Eq, Ord, Ix, Bounded, Enum, Read, Show)
data BufferMode  =  NoBuffering | LineBuffering
                 |  BlockBuffering (Maybe Int)
                    deriving (Eq, Ord, Read, Show)
data SeekMode    =  AbsoluteSeek | RelativeSeek | SeekFromEnd
                    deriving (Eq, Ord, Ix, Bounded, Enum, Read, Show)

stdin, stdout, stderr :: Handle

openFile              :: FilePath -> IOMode -> IO Handle
hClose                :: Handle -> IO ()
```

```
hFileSize             :: Handle -> IO Integer
hIsEOF                :: Handle -> IO Bool
isEOF                 :: IO Bool
isEOF                 =  hIsEOF stdin

hSetBuffering         :: Handle  -> BufferMode -> IO ()
hGetBuffering         :: Handle  -> IO BufferMode
hFlush                :: Handle -> IO ()
hGetPosn              :: Handle -> IO HandlePosn
hSetPosn              :: HandlePosn -> IO ()
hSeek                 :: Handle -> SeekMode -> Integer -> IO ()

hWaitForInput         :: Handle -> Int -> IO Bool
hReady                :: Handle -> IO Bool
hReady h              =  hWaitForInput h 0
hGetChar              :: Handle -> IO Char
hGetLine              :: Handle -> IO String
hLookAhead            :: Handle -> IO Char
hGetContents          :: Handle -> IO String
hPutChar              :: Handle -> Char -> IO ()
hPutStr               :: Handle -> String -> IO ()
hPutStrLn             :: Handle -> String -> IO ()
hPrint                :: Show a => Handle -> a -> IO ()

hIsOpen               :: Handle -> IO Bool
hIsClosed             :: Handle -> IO Bool
hIsReadable           :: Handle -> IO Bool
hIsWritable           :: Handle -> IO Bool
hIsSeekable           :: Handle -> IO Bool

isAlreadyExistsError  :: IOError -> Bool
isDoesNotExistError   :: IOError -> Bool
isAlreadyInUseError   :: IOError -> Bool
isFullError           :: IOError -> Bool
isEOFError            :: IOError -> Bool
isIllegalOperation    :: IOError -> Bool
isPermissionError     :: IOError -> Bool
isUserError           :: IOError -> Bool

ioeGetErrorString     :: IOError -> String
ioeGetHandle          :: IOError -> Maybe Handle
ioeGetFileName        :: IOError -> Maybe FilePath

try                   :: IO a -> Either IOError a
bracket               :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket_              :: IO a -> (a -> IO b) -> IO c -> IO c
```

The monadic I/O system used in Haskell is described by the Haskell language report. Commonly used I/O functions such as `print` are part of the standard prelude and need not be explicitly imported. This library contain more advanced I/O features. Some related operations on file systems are contained in the `Directory` library.

11.1 I/O Errors

Errors of type `IOError` are used by the I/O monad. This is an abstract type; the library provides functions to interrogate and construct values in `IOError`:

- `isAlreadyExistsError` – the operation failed because one of its arguments already exists.
- `isDoesNotExistError` – the operation failed because one of its arguments does not exist.
- `isAlreadyInUseError` – the operation failed because one of its arguments is a single-use resource, which is already being used (for example, opening the same file twice for writing might give this error).
- `isFullError` – the operation failed because the device is full.
- `isEOFError` – the operation failed because the end of file has been reached.
- `isIllegalOperation` – the operation is not possible.
- `isPermissionError` – the operation failed because the user does not have sufficient operating system privilege to perform that operation.
- `isUserError` – a programmer-defined error value has been raised using `fail`.

All these functions return return a `Bool`, which is `True` if its argument is the corresponding kind of error, and `False` otherwise.

Any computation which returns an `IO` result may fail with `isIllegalOperationError`. Additional errors which could be raised by an implementation are listed after the corresponding operation. In some cases, an implementation will not be able to distinguish between the possible error causes. In this case it should return `isIllegalOperationError`.

Three additional functions are provided to obtain information about an error value. These are `ioeGetHandle` which returns `Just` *hdl* if the error value refers to handle *hdl* and `Nothing` otherwise; `ioeGetFileName` which returns `Just` *name* if the error value refers to file *name*, and `Nothing` otherwise; and `ioeGetErrorString` which returns a string. For "user" errors (those which are raised using `fail`), the string returned by `ioeGetErrorString` is the argument that was passed to `fail`; for all other errors, the string is implementation-dependent.

The `try` function returns an error in a computation explicitly using the `Either` type.

The `bracket` function captures a common allocate, compute, deallocate idiom in which the deallocation step must occur even in the case of an error during computation. This is similar to try-catch-finally in Java.

```
module IO where

-- Just provide an implementation of the system-indendent
-- actions that IO exports.

try            :: IO a -> IO (Either IOError a)
try f          =  catch (do r <- f
                            return (Right r))
                        (return . Left)

bracket        :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket before after m = do
        x  <- before
        rs <- try (m x)
        after x
        case rs of
           Right r -> return r
           Left  e -> ioError e

-- variant of the above where middle computation doesn't want x
bracket_        :: IO a -> (a -> IO b) -> IO c -> IO c
bracket_ before after m = do
         x  <- before
         rs <- try m
         after x
         case rs of
            Right r -> return r
            Left  e -> ioError e
```

11.2 Files and Handles

Haskell interfaces to the external world through an abstract *file system*. This file system is a collection of named *file system objects*, which may be organised in *directories* (see `Directory`). In some implementations, directories may themselves be file system objects and could be entries in other directories. For simplicity, any non-directory file system object is termed a *file*, although it could in fact be a communication channel, or any other object recognised by the operating system. *Physical files* are persistent, ordered files, and normally reside on disk.

File and directory names are values of type `String`, whose precise meaning is operating system dependent. Files can be opened, yielding a handle which can then be used to operate on the contents of that file.

Haskell defines operations to read and write characters from and to files, represented by values of type `Handle`. Each value of this type is a *handle*: a record used by the Haskell run-time system to *manage* I/O with file system objects. A handle has at least the following properties:

- whether it manages input or output or both;
- whether it is *open*, *closed* or *semi-closed*;
- whether the object is seekable;
- whether buffering is disabled, or enabled on a line or block basis;
- a buffer (whose length may be zero).

Most handles will also have a current I/O position indicating where the next input or output operation will occur. A handle is *readable* if it manages only input or both input and output; likewise, it is *writable* if it manages only output or both input and output. A handle is *open* when first allocated. Once it is closed it can no longer be used for either input or output, though an implementation cannot re-use its storage while references remain to it. Handles are in the `Show` and `Eq` classes. The string produced by showing a handle is system dependent; it should include enough information to identify the handle for debugging. A handle is equal according to `==` only to itself; no attempt is made to compare the internal state of different handles for equality.

11.2.1 Semi-Closed Handles

The operation `hGetContents` puts a handle *hdl* into an intermediate state, *semi-closed*. In this state, *hdl* is effectively closed, but items are read from *hdl* on demand and accumulated in a special stream returned by `hGetContents` *hdl*.

Any operation except for `hClose` that fails because a handle is closed, also fails if a handle is semi-closed. A semi-closed handle becomes closed:

- if `hClose` is applied to it;
- if an I/O error occurs when reading an item from the file item from the stream;
- or once the entire contents of the file has been read.

Once a semi-closed handle becomes closed, the contents of the associated stream becomes fixed, and is the list of those items which were successfully read from that handle. Any I/O errors encountered while a handle is semi-closed are simply discarded.

11.2.2 Standard Handles

Three handles are allocated during program initialisation. The first two (`stdin` and `stdout`) manage input or output from the Haskell program's standard input or output channel respectively. The third (`stderr`) manages output to the standard error channel. These handles are initially open.

11.3 Opening and Closing Files

11.3.1 Opening Files

Computation `openFile` *file mode* allocates and returns a new, open handle to manage the file *file*. It manages input if *mode* is `ReadMode`, output if *mode* is `WriteMode` or `AppendMode`, and both input and output if mode is `ReadWriteMode`.

If the file does not exist and it is opened for output, it should be created as a new file. If *mode* is `WriteMode` and the file already exists, then it should be truncated to zero length. Some operating systems delete empty files, so there is no guarantee that the file will exist following an `openFile` with *mode* `WriteMode` unless it is subsequently written to successfully. The handle is positioned at the end of the file if *mode* is `AppendMode`, and otherwise at the beginning (in which case its internal I/O position is 0). The initial buffer mode is implementation-dependent.

If `openFile` fails on a file opened for output, the file may still have been created if it did not already exist.

Implementations should enforce as far as possible, locally to the Haskell process, multiple-reader single-writer locking on files. Thus there may either be many handles on the same file which manage input, or just one handle on the file which manages output. If any open or semi-closed handle is managing a file for output, no new handle can be allocated for that file. If any open or semi-closed handle is managing a file for input, new handles can only be allocated if they do not manage output. Whether two files are the same is implementation-dependent, but they should normally be the same if they have the same absolute path name and neither has been renamed, for example.

Error reporting: the `openFile` computation may fail with `isAlreadyInUseError` if the file is already open and cannot be reopened; `isDoesNotExistError` if the file does not exist; or `isPermissionError` if the user does not have permission to open the file.

11.3.2 Closing Files

Computation `hClose` *hdl* makes handle *hdl* closed. Before the computation finishes, if *hdl* is writable its buffer is flushed as for `hFlush`. If the operation fails for any reason, any further operations on the handle will still fail as if *hdl* had been successfully closed.

11.4 Determining the Size of a File

For a handle *hdl* which is attached to a physical file, `hFileSize` *hdl* returns the size of that file in 8-bit bytes (≥ 0).

11.4.1 Detecting the End of Input

For a readable handle *hdl*, computation `hIsEOF` *hdl* returns `True` if no further input can be taken from *hdl*; for a handle attached to a physical file this means that the current I/O position is equal to the length of the file. Otherwise, it returns `False`. The computation `isEOF` is identical, except that it works only on `stdin`.

11.4.2 Buffering Operations

Three kinds of buffering are supported: line-buffering, block-buffering or no-buffering. These modes have the following effects. For output, items are written out from the internal buffer according to the buffer mode:

- **line-buffering:** the entire buffer is written out whenever a newline is output, the buffer overflows, a flush is issued, or the handle is closed.
- **block-buffering:** the entire buffer is written out whenever it overflows, a flush is issued, or the handle is closed.
- **no-buffering:** output is written immediately, and never stored in the buffer.

The buffer is emptied as soon as it has been written out.

Similarly, input occurs according to the buffer mode for handle *hdl*.

- **line-buffering:** when the buffer for *hdl* is not empty, the next item is obtained from the buffer; otherwise, when the buffer is empty, characters are read into the buffer until the next newline character is encountered or the buffer is full. No characters are available until the newline character is available or the buffer is full.
- **block-buffering:** when the buffer for *hdl* becomes empty, the next block of data is read into the buffer.
- **no-buffering:** the next input item is read and returned.

For most implementations, physical files will normally be block-buffered and terminals will normally be line-buffered.

Computation `hSetBuffering` *hdl mode* sets the mode of buffering for handle *hdl* on subsequent reads and writes.

- If *mode* is `LineBuffering`, line-buffering is enabled if possible.
- If *mode* is `BlockBuffering` *size*, then block-buffering is enabled if possible. The size of the buffer is n items if *size* is `Just` n and is otherwise implementation-dependent.
- If *mode* is `NoBuffering`, then buffering is disabled if possible.

If the buffer mode is changed from `BlockBuffering` or `LineBuffering` to `NoBuffering`, then

- if *hdl* is writable, the buffer is flushed as for `hFlush`;
- if *hdl* is not writable, the contents of the buffer is discarded.

Error reporting: the `hSetBuffering` computation may fail with `isPermissionError` if the handle has already been used for reading or writing and the implementation does not allow the buffering mode to be changed.

Computation `hGetBuffering` *hdl* returns the current buffering mode for *hdl*.

The default buffering mode when a handle is opened is implementation-dependent and may depend on the file system object which is attached to that handle.

11.4.3 Flushing Buffers

Computation `hFlush` *hdl* causes any items buffered for output in handle *hdl* to be sent immediately to the operating system.

Error reporting: the `hFlush` computation may fail with: `isFullError` if the device is full; `isPermissionError` if a system resource limit would be exceeded. It is unspecified whether the characters in the buffer are discarded or retained under these circumstances.

11.5 Repositioning Handles

11.5.1 Revisiting an I/O Position

Computation `hGetPosn` *hdl* returns the current I/O position of *hdl* as a value of the abstract type `HandlePosn`. If a call to `hGetPosn` h returns a position p, then computation `hSetPosn` p sets the position of h to the position it held at the time of the call to `hGetPosn`.

Error reporting: the `hSetPosn` computation may fail with: `isPermissionError` if a system resource limit would be exceeded.

11.5.2 Seeking to a new Position

Computation `hSeek` *hdl mode i* sets the position of handle *hdl* depending on *mode*. If *mode* is:

- `AbsoluteSeek`: the position of *hdl* is set to *i*.
- `RelativeSeek`: the position of *hdl* is set to offset *i* from the current position.
- `SeekFromEnd`: the position of *hdl* is set to offset *i* from the end of the file.

The offset is given in terms of 8-bit bytes.

If *hdl* is block- or line-buffered, then seeking to a position which is not in the current buffer will first cause any items in the output buffer to be written to the device, and then cause the input buffer to be discarded. Some handles may not be seekable (see `hIsSeekable`), or only support a subset of the possible positioning operations (for instance, it may only be possible to seek to the end of a tape, or to a positive offset from the beginning or current position). It is not possible to set a negative I/O position, or for a physical file, an I/O position beyond the current end-of-file.

Error reporting: the `hSeek` computation may fail with: `isPermissionError` if a system resource limit would be exceeded.

11.6 Handle Properties

The functions `hIsOpen`, `hIsClosed`, `hIsReadable`, `hIsWritable` and `hIsSeekable` return information about the properties of a handle. Each of these returns `True` if the handle has the specified property, and `False` otherwise.

11.7 Text Input and Output

Here we define a standard set of input operations for reading characters and strings from text files, using handles. Many of these functions are generalizations of Prelude functions. I/O in the Prelude generally uses `stdin` and `stdout`; here, handles are explicitly specified by the I/O operation.

11.7.1 Checking for Input

Computation `hWaitForInput` *hdl* *t* waits until input is available on handle *hdl*. It returns `True` as soon as input is available on *hdl*, or `False` if no input is available within *t* milliseconds.

Computation `hReady` *hdl* indicates whether at least one item is available for input from handle *hdl*.

Computation `hGetChar` *hdl* reads a character from the file or channel managed by *hdl*.

Computation `hGetLine` *hdl* reads a line from the file or channel managed by *hdl*, similar to `getLine` in the Prelude.

Error reporting: the `hWaitForInput`, `hReady` and `hGetChar` computations may fail with: `isEOFError` if the end of file has been reached.

11.7.2 Reading Ahead

Computation `hLookAhead` *hdl* returns the next character from handle *hdl* without removing it from the input buffer, blocking until a character is available.

Error reporting: the `hLookahead` computation may fail with: `isEOFError` if the end of file has been reached.

Computation `hGetContents` *hdl* returns the list of characters corresponding to the unread portion of the channel or file managed by *hdl*, which is made semi-closed.

Error reporting: the `hGetContents` computation may fail with: `isEOFError` if the end of file has been reached.

Computation `hPutChar` *hdl c* writes the character *c* to the file or channel managed by *hdl*. Characters may be buffered if buffering is enabled for *hdl*.

Computation `hPutStr` *hdl s* writes the string *s* to the file or channel managed by *hdl*.

Computation `hPrint` *hdl t* writes the string representation of *t* given by the `shows` function to the file or channel managed by *hdl* and appends a newline.

Error reporting: the `hPutChar`, `hPutStr` and `hPrint` computations may fail with: `isFull-Error` if the device is full; or `isPermissionError` if another system resource limit would be exceeded.

11.8 Examples

Here are some simple examples to illustrate Haskell I/O.

11.8.1 Summing Two Numbers

This program reads and sums two `Integers`.

```
import IO

main = do
         hSetBuffering stdout NoBuffering
         putStr   "Enter an integer: "
         x1 <- readNum
         putStr   "Enter another integer: "
         x2 <- readNum
         putStr  ("Their sum is " ++ show (x1+x2) ++ "\n")
       where readNum :: IO Integer
             readNum =  do { line <- getLine; readIO line }
```

11.8.2 Copying Files

A simple program to create a copy of a file, with all lower-case characters translated to upper-case. This program will not allow a file to be copied to itself. This version uses character-level I/O. Note that exactly two arguments must be supplied to the program.

```
import IO
import System

main = do
         [f1,f2] <- getArgs
         h1 <- openFile f1 ReadMode
         h2 <- openFile f2 WriteMode
         copyFile h1 h2
         hClose h1
         hClose h2

copyFile h1 h2 = do
                   eof <- hIsEOF h1
                   if eof then return () else
                      do
                        c <- hGetChar h1
                        hPutChar h2 (toUpper c)
                        copyFile h1 h2
```

An equivalent but much shorter version, using string I/O is:

```
import System

main = do
         [f1,f2] <- getArgs
         s <- readFile f1
         writeFile f2 (map toUpper s)
```

12 Directory Functions

```
module Directory (
    Permissions,
    readable, writable, executable, searchable,
    createDirectory, removeDirectory, removeFile,
    renameDirectory, renameFile, getDirectoryContents,
    getCurrentDirectory, setCurrentDirectory,
    doesFileExist, doesDirectoryExist,
    getPermissions, setPermissions,
    getModificationTime ) where

import Time ( ClockTime )

data Permissions = ...  -- Abstract

instance Eq   Permissions where ...
instance Ord  Permissions where ...
instance Read Permissions where ...
instance Show Permissions where ...

readable, writable, executable, searchable :: Permissions -> Bool

createDirectory        :: FilePath -> IO ()
removeDirectory        :: FilePath -> IO ()
removeFile             :: FilePath -> IO ()
renameDirectory        :: FilePath -> FilePath -> IO ()
renameFile             :: FilePath -> FilePath -> IO ()

getDirectoryContents   :: FilePath -> IO [FilePath]
getCurrentDirectory    :: IO FilePath
setCurrentDirectory    :: FilePath -> IO ()

doesFileExist          :: FilePath -> IO Bool
doesDirectoryExist     :: FilePath -> IO Bool

getPermissions         :: FilePath -> IO Permissions
setPermissions         :: FilePath -> Permissions -> IO ()

getModificationTime    :: FilePath -> IO ClockTime
```

These functions operate on directories in the file system.

Any `Directory` operation could raise an `isIllegalOperationError`, as described in Section 11.1; all other permissible errors are described below. Note that, in particular, if an implementation does not support an operation it should raise an `isIllegalOperationError`.

A directory contains a series of entries, each of which is a named reference to a file system object (file, directory etc.). Some entries may be hidden, inaccessible, or have some administrative function (for instance, "." or ".." under POSIX), but all such entries are considered to form part of the directory contents. Entries in sub-directories are not, however, considered to form part of the directory contents. Although there may be file system objects other than files and directories, this library does not distinguish between physical files and other non-directory objects. All such objects should therefore be treated as if they are files.

Each file system object is referenced by a *path*. There is normally at least one absolute path to each file system object. In some operating systems, it may also be possible to have paths which are relative to the current directory.

Computation `createDirectory` *dir* creates a new directory *dir* which is initially empty, or as near to empty as the operating system allows.

Error reporting: the `createDirectory` computation may fail with: `isPermissionError` if the user is not permitted to create the directory; `isAlreadyExistsError` if the directory already exists.

Computation `removeDirectory` *dir* removes an existing directory *dir*. The implementation may specify additional constraints which must be satisfied before a directory can be removed (for instance, the directory has to be empty, or may not be in use by other processes). It is not legal for an implementation to partially remove a directory unless the entire directory is removed. A conformant implementation need not support directory removal in all situations (for instance, removal of the root directory).

Computation `removeFile` *file* removes the directory entry for an existing file *file*, where *file* is not itself a directory. The implementation may specify additional constraints which must be satisfied before a file can be removed (for instance, the file may not be in use by other processes).

Error reporting: the `removeDirectory` and `removeFile` computations may fail with: `isPermissionError` if the user is not permitted to remove the file/directory; or `isDoesNotExistError` if the file/directory does not exist.

Computation `renameDirectory` *old new* changes the name of an existing directory from *old* to *new*. If the *new* directory already exists, it is atomically replaced by the *old* directory. If the *new* directory is neither the *old* directory nor an alias of the *old* directory, it is removed as if by `removeDirectory`. A conformant implementation need not support renaming directories in all situations (for instance, renaming to an existing directory, or across different physical devices), but the constraints must be documented.

Computation `renameFile` *old new* changes the name of an existing file system object from *old* to *new*. If the *new* object already exists, it is atomically replaced by the *old* object. Neither path may refer to an existing directory. A conformant implementation need not support renaming files in all situations (for instance, renaming across different physical devices), but the constraints must be documented.

Error reporting: the `renameDirectory` and `renameFile` computations may fail with: `isPermissionError` if the user is not permitted to rename the file/directory, or if either argument to `renameFile` is a directory; or `isDoesNotExistError` if the file/directory does not exist.

Computation `getDirectoryContents` *dir* returns a list of *all* entries in *dir*.

If the operating system has a notion of current directories, `getCurrentDirectory` returns an absolute path to the current directory of the calling process.

Error reporting: the `getDirectoryContents` and `getCurrentDirectory` computations may fail with: `isPermissionError` if the user is not permitted to access the directory; or `isDoesNotExistError` if the directory does not exist.

If the operating system has a notion of current directories, `setCurrentDirectory` *dir* changes the current directory of the calling process to *dir*.

Error reporting: the `setCurrentDirectory` computation may fail with: `isPermissionError` if the user is not permitted to change directory to that specified; or `isDoesNotExistError` if the directory does not exist.

The `Permissions` type is used to record whether certain operations are permissible on a file/directory. `getPermissions` and `setPermissions` get and set these permissions, respectively. Permissions apply both to files and directories. For directories, the `executable` field will be `False`, and for files the `searchable` field will be `False`. Note that directories may be searchable without being readable, if permission has been given to use them as part of a path, but not to examine the directory contents.

Note that to change some, but not all permissions, a construct on the following lines must be used.

```
makeReadable f = do
                     p <- getPermissions f
                     setPermissions f (p {readable = True})
```

The operation `doesDirectoryExist` returns `True` if the argument file exists and is a directory, and `False` otherwise. The operation `doesFileExist` returns `True` if the argument file exists and is not a directory, and `False` otherwise.

The `getModificationTime` operation returns the clock time at which the file/directory was last modified.

Error reporting: the `get(set)Permissions`, `doesFile(Directory)Exist`, and `getModificationTime` computations may fail with: `isPermissionError` if the user is not permitted to access the appropriate information; or `isDoesNotExistError` if the file/directory does not exist. The `setPermissions` computation may also fail with: `isPermissionError` if the user is not permitted to change the permission for the specified file or directory; or `isDoesNotExistError` if the file/directory does not exist.

13 System Functions

```
module System (
    ExitCode(ExitSuccess,ExitFailure),
    getArgs, getProgName, getEnv, system, exitWith, exitFailure
  ) where

data ExitCode = ExitSuccess | ExitFailure Int
                deriving (Eq, Ord, Read, Show)

getArgs                 :: IO [String]
getProgName             :: IO String
getEnv                  :: String -> IO String
system                  :: String -> IO ExitCode
exitWith                :: ExitCode -> IO a
exitFailure             :: IO a
```

This library describes the interaction of the program with the operating system.

Any `System` operation could raise an `isIllegalOperationError`, as described in Section 11.1; all other permissible errors are described below. Note that, in particular, if an implementation does not support an operation it must raise an `isIllegalOperationError`.

The `ExitCode` type defines the exit codes that a program can return. `ExitSuccess` indicates successful termination; and `ExitFailure` *code* indicates program failure with value *code*. The exact interpretation of *code* is operating-system dependent. In particular, some values of *code* may be prohibited (for instance, 0 on a POSIX-compliant system).

Computation `getArgs` returns a list of the program's command line arguments (not including the program name). Computation `getProgName` returns the name of the program as it was invoked. Computation `getEnv` *var* returns the value of the environment variable *var*. If variable *var* is undefined, the `isDoesNotExistError` exception is raised.

Computation `system` *cmd* returns the exit code produced when the operating system processes the command *cmd*.

Computation `exitWith` *code* terminates the program, returning *code* to the program's caller. Before the program terminates, any open or semi-closed handles are first closed. The caller may interpret the return code as it wishes, but the program should return `ExitSuccess` to mean normal completion, and `ExitFailure` *n* to mean that the program encountered a problem from which it could not recover. The value `exitFailure` is equal to `exitWith (ExitFailure` *exitfail*`)`, where *exitfail* is implementation-dependent. `exitWith` bypasses the error handling in the I/O monad and cannot be intercepted by `catch`.

If a program terminates as a result of calling `error` or because its value is otherwise determined to be $\perp$, then it is treated identically to the computation `exitFailure`. Otherwise, if any program p terminates without calling `exitWith` explicitly, it is treated identically to the computation

```
(p >> exitWith ExitSuccess) `catch` \ _ -> exitFailure
```

14 Dates and Times

```
module Time (
        ClockTime,
        Month(January,February,March,April,May,June,
              July,August,September,October,November,December),
        Day(Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday),
        CalendarTime(CalendarTime, ctYear, ctMonth, ctDay, ctHour, ctMin,
                     ctPicosec, ctWDay, ctYDay, ctTZName, ctTZ, ctIsDST),
        TimeDiff(TimeDiff, tdYear, tdMonth, tdDay, tdHour,
                 tdMin, tdSec, tdPicosec),
        getClockTime, addToClockTime, diffClockTimes,
        toCalendarTime, toUTCTime, toClockTime,
        calendarTimeToString, formatCalendarTime ) where

import Ix(Ix)

data ClockTime = ...                    -- Implementation-dependent
instance Ord  ClockTime where ...
instance Eq   ClockTime where ...

data Month =  January   | February | March    | April
           |  May       | June     | July     | August
           |  September | October  | November | December
           deriving (Eq, Ord, Enum, Bounded, Ix, Read, Show)

data Day   =  Sunday | Monday  | Tuesday  | Wednesday | Thursday
           |  Friday | Saturday
           deriving (Eq, Ord, Enum, Bounded, Ix, Read, Show)

data CalendarTime = CalendarTime {
                ctYear                          :: Int,
                ctMonth                         :: Month,
                ctDay, ctHour, ctMin, ctSec     :: Int,
                ctPicosec                       :: Integer,
                ctWDay                          :: Day,
                ctYDay                          :: Int,
                ctTZName                        :: String,
                ctTZ                            :: Int,
                ctIsDST                         :: Bool
        } deriving (Eq, Ord, Read, Show)

data TimeDiff = TimeDiff {
                tdYear, tdMonth, tdDay, tdHour, tdMin, tdSec :: Int,
                tdPicosec                                    :: Integer
        } deriving (Eq, Ord, Read, Show)
```

```
-- Functions on times
getClockTime         :: IO ClockTime

addToClockTime       :: TimeDiff  -> ClockTime -> ClockTime
diffClockTimes       :: ClockTime -> ClockTime -> TimeDiff

toCalendarTime       :: ClockTime    -> IO CalendarTime
toUTCTime            :: ClockTime    -> CalendarTime
toClockTime          :: CalendarTime -> ClockTime
calendarTimeToString :: CalendarTime -> String
formatCalendarTime   :: TimeLocale -> String -> CalendarTime -> String
```

The `Time` library provides standard functionality for clock times, including timezone information. It follows RFC 1129 in its use of Coordinated Universal Time (UTC).

`ClockTime` is an abstract type, used for the system's internal clock time. Clock times may be compared directly or converted to a calendar time `CalendarTime` for I/O or other manipulations. `CalendarTime` is a user-readable and manipulable representation of the internal `ClockTime` type. The numeric fields have the following ranges.

Value	Range	Comments
ctYear	-maxInt ... maxInt	Pre-Gregorian dates are inaccurate
ctDay	1 ... 31	
ctHour	0 ... 23	
ctMin	0 ... 59	
ctSec	0 ... 61	Allows for two Leap Seconds
ctPicosec	0 ... $(10^{12}) - 1$	
ctYDay	0 ... 365	364 in non-Leap years
ctTZ	-89999 ... 89999	Variation from UTC in seconds

The *ctTZName* field is the name of the time zone. The *ctIsDST* field is `True` if Daylight Savings Time would be in effect, and `False` otherwise. The `TimeDiff` type records the difference between two clock times in a user-readable way.

Function `getClockTime` returns the current time in its internal representation. The expression `addToClockTime` *d t* adds a time difference *d* and a clock time *t* to yield a new clock time. The difference *d* may be either positive or negative. The expression `diffClockTimes` *t1 t2* returns the difference between two clock times *t1* and *t2* as a `TimeDiff`.

Function `toCalendarTime` *t* converts *t* to a local time, modified by the timezone and daylight savings time settings in force at the time of conversion. Because of this dependence on the local environment, `toCalendarTime` is in the `IO` monad.

Function `toUTCTime` *t* converts *t* into a `CalendarTime` in standard UTC format.

`toClockTime` l converts l into the corresponding internal `ClockTime` ignoring the contents of the *ctWDay*, *ctYDay*, *ctTZName*, and *ctIsDST* fields.

Function `calendarTimeToString` formats calendar times using local conventions and a formatting string.

14.1 Library `Time`

```
module Time (
        ClockTime,
        Month(January,February,March,April,May,June,
              July,August,September,October,November,December),
        Day(Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday),
        CalendarTime(CalendarTime, ctYear, ctMonth, ctDay, ctHour, ctMin,
                     ctPicosec, ctWDay, ctYDay, ctTZName, ctTZ, ctIsDST),
        TimeDiff(TimeDiff, tdYear, tdMonth, tdDay,
                 tdHour, tdMin, tdSec, tdPicosec),
        getClockTime, addToClockTime, diffClockTimes,
        toCalendarTime, toUTCTime, toClockTime,
        calendarTimeToString, formatCalendarTime ) where

import Ix(Ix)
import Locale(TimeLocale(..),defaultTimeLocale)
import Char ( intToDigit )

data ClockTime = ...                    -- Implementation-dependent
instance Ord  ClockTime where ...
instance Eq   ClockTime where ...

data Month =  January   | February | March    | April
           |  May       | June     | July     | August
           |  September | October  | November | December
           deriving (Eq, Ord, Enum, Bounded, Ix, Read, Show)

data Day   =  Sunday | Monday  | Tuesday  | Wednesday | Thursday
           |  Friday | Saturday
           deriving (Eq, Ord, Enum, Bounded, Ix, Read, Show)
```

```
data CalendarTime = CalendarTime {
                ctYear                          :: Int,
                ctMonth                         :: Month,
                ctDay, ctHour, ctMin, ctSec     :: Int,
                ctPicosec                       :: Integer,
                ctWDay                          :: Day,
                ctYDay                          :: Int,
                ctTZName                        :: String,
                ctTZ                            :: Int,
                ctIsDST                         :: Bool
        } deriving (Eq, Ord, Read, Show)

data TimeDiff = TimeDiff {
                tdYear, tdMonth, tdDay, tdHour, tdMin, tdSec :: Int,
                tdPicosec                                    :: Integer
        } deriving (Eq, Ord, Read, Show)

getClockTime            :: IO ClockTime
getClockTime            = ...           -- Implementation-dependent

addToClockTime          :: TimeDiff     -> ClockTime -> ClockTime
addToClockTime td ct    =  ...          -- Implementation-dependent

diffClockTimes          :: ClockTime    -> ClockTime -> TimeDiff
diffClockTimes ct1 ct2  =  ...          -- Implementation-dependent

toCalendarTime          :: ClockTime    -> IO CalendarTime
toCalendarTime ct       =  ...          -- Implementation-dependent

toUTCTime               :: ClockTime    -> CalendarTime
toUTCTime ct            =  ...          -- Implementation-dependent

toClockTime             :: CalendarTime -> ClockTime
toClockTime cal         =  ...          -- Implementation-dependent

calendarTimeToString    :: CalendarTime -> String
calendarTimeToString    =  formatCalendarTime defaultTimeLocale "%c"
```

```
formatCalendarTime :: TimeLocale -> String -> CalendarTime -> String
formatCalendarTime l fmt ct@(CalendarTime year mon day hour min sec sdec
                                          wday yday tzname _ _) =
        doFmt fmt
  where doFmt ('%':c:cs) = decode c ++ doFmt cs
        doFmt (c:cs) = c : doFmt cs
        doFmt "" = ""

        to12 :: Int -> Int
        to12 h = let h' = h `mod` 12 in if h' == 0 then 12 else h'

        decode 'A' = fst (wDays l  !! fromEnum wday)
        decode 'a' = snd (wDays l  !! fromEnum wday)
        decode 'B' = fst (months l !! fromEnum mon)
        decode 'b' = snd (months l !! fromEnum mon)
        decode 'h' = snd (months l !! fromEnum mon)
        decode 'C' = show2 (year `quot` 100)
        decode 'c' = doFmt (dateTimeFmt l)
        decode 'D' = doFmt "%m/%d/%y"
        decode 'd' = show2 day
        decode 'e' = show2' day
        decode 'H' = show2 hour
        decode 'I' = show2 (to12 hour)
        decode 'j' = show3 yday
        decode 'k' = show2' hour
        decode 'l' = show2' (to12 hour)
        decode 'M' = show2 min
        decode 'm' = show2 (fromEnum mon+1)
        decode 'n' = "\n"
        decode 'p' = (if hour < 12 then fst else snd) (amPm l)
        decode 'R' = doFmt "%H:%M"
        decode 'r' = doFmt (time12Fmt l)
        decode 'T' = doFmt "%H:%M:%S"
        decode 't' = "\t"
        decode 'S' = show2 sec
        decode 's' = ...                -- Implementation-dependent
        decode 'U' = show2 ((yday + 7 - fromEnum wday) `div` 7)
        decode 'u' = show (let n = fromEnum wday in
                           if n == 0 then 7 else n)
        decode 'V' =
            let (week, days) =
                   (yday + 7 - if fromEnum wday > 0 then
                               fromEnum wday - 1 else 6) `divMod` 7
            in  show2 (if days >= 4 then
```

```
                        week+1
                     else if week == 0 then 53 else week)
        decode 'W' =
            show2 ((yday + 7 - if fromEnum wday > 0 then
                                  fromEnum wday - 1 else 6) `div` 7)
        decode 'w' = show (fromEnum wday)
        decode 'X' = doFmt (timeFmt l)
        decode 'x' = doFmt (dateFmt l)
        decode 'Y' = show year
        decode 'y' = show2 (year `rem` 100)
        decode 'Z' = tzname
        decode '%' = "%"
        decode c   = [c]

show2, show2', show3 :: Int -> String
show2 x = [intToDigit (x `quot` 10), intToDigit (x `rem` 10)]

show2' x = if x < 10 then [ ' ', intToDigit x] else show2 x

show3 x = intToDigit (x `quot` 100) : show2 (x `rem` 100)
```

15 Locale

```
module Locale(TimeLocale(..), defaultTimeLocale) where

data TimeLocale = TimeLocale {
        wDays  :: [(String, String)],   -- full and abbreviated week days
        months :: [(String, String)],   -- full and abbreviated months
        amPm   :: (String, String),     -- AM/PM symbols
        dateTimeFmt, dateFmt,           -- formatting strings
          timeFmt, time12Fmt :: String
        } deriving (Eq, Ord, Show)

defaultTimeLocale :: TimeLocale
```

The `Locale` library provides the ability to adapt to local conventions. At present, it supports only time and date information as used by `calendarTimeToString` from the `Time` library.

15.1 Library Locale

```
module Locale(TimeLocale(..), defaultTimeLocale) where

data TimeLocale = TimeLocale {
        wDays  :: [(String, String)],   -- full and abbreviated week days
        months :: [(String, String)],   -- full and abbreviated months
        amPm   :: (String, String),     -- AM/PM symbols
        dateTimeFmt, dateFmt,           -- formatting strings
          timeFmt, time12Fmt :: String
        } deriving (Eq, Ord, Show)

defaultTimeLocale :: TimeLocale
defaultTimeLocale =  TimeLocale {
        wDays  = [("Sunday",   "Sun"),  ("Monday",    "Mon"),
                  ("Tuesday",  "Tue"),  ("Wednesday", "Wed"),
                  ("Thursday", "Thu"),  ("Friday",    "Fri"),
                  ("Saturday", "Sat")],

        months = [("January",   "Jan"), ("February",  "Feb"),
                  ("March",     "Mar"), ("April",     "Apr"),
                  ("May",       "May"), ("June",      "Jun"),
                  ("July",      "Jul"), ("August",    "Aug"),
                  ("September", "Sep"), ("October",   "Oct"),
                  ("November",  "Nov"), ("December",  "Dec")],

        amPm = ("AM", "PM"),
        dateTimeFmt = "%a %b %e %H:%M:%S %Z %Y",
        dateFmt = "%m/%d/%y",
        timeFmt = "%H:%M:%S",
        time12Fmt = "%I:%M:%S %p"
        }
```

16 CPU Time

```
module CPUTime ( getCPUTime ) where

getCPUTime        :: IO Integer
cpuTimePrecision  :: Integer
```

Computation `getCPUTime` returns the number of picoseconds of CPU time used by the current program. The precision of this result is given by `cpuTimePrecision`. This is the smallest measurable difference in CPU time that the implementation can record, and is given as an integral number of picoseconds.

17 Random Numbers

```
module Random (
        RandomGen(next, split),
        StdGen, mkStdGen,
        Random( random,   randomR,
                randoms,  randomRs,
                randomIO, randomRIO ),
        getStdRandom, getStdGen, setStdGen, newStdGen
  ) where

---------------- The RandomGen class ------------------------

class RandomGen g where
  next  :: g  -> (Int, g)
  split :: g -> (g, g)

---------------- A standard instance of RandomGen -----------
data StdGen = ...       -- Abstract

instance RandomGen StdGen where ...
instance Read      StdGen where ...
instance Show      StdGen where ...

mkStdGen :: Int -> StdGen

---------------- The Random class ---------------------------
class Random a where
   randomR :: RandomGen g => (a, a) -> g -> (a, g)
   random  :: RandomGen g => g -> (a, g)

   randomRs :: RandomGen g => (a, a) -> g -> [a]
   randoms  :: RandomGen g => g -> [a]

   randomRIO :: (a,a) -> IO a
   randomIO  :: IO a

instance Random Int     where ...
instance Random Integer where ...
instance Random Float   where ...
instance Random Double  where ...
instance Random Bool    where ...
instance Random Char    where ...

---------------- The global random generator ----------------
newStdGen    :: IO StdGen
setStdGen    :: StdGen -> IO ()
getStdGen    :: IO StdGen
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

The Random library deals with the common task of pseudo-random number generation. The library makes it possible to generate repeatable results, by starting with a specified initial random number generator; or to get different results on each run by using the system-initialised generator, or by supplying a seed from some other source.

The library is split into two layers:

- A core *random number generator* provides a supply of bits. The class RandomGen provides a common interface to such generators.
- The class Random provides a way to extract particular values from a random number generator. For example, the Float instance of Random allows one to generate random values of type Float.

17.1 The RandomGen class, and the StdGen generator

The class RandomGen provides a common interface to random number generators.

```
class RandomGen g where
  next  :: g  -> (Int, g)
  split :: g -> (g, g)
```

- The next operation allows one to extract at least 30 bits (one Int's worth) from the generator, returning a new generator as well. The integer returned may be positive or negative.
- The split operation allows one to obtain two distinct random number generators. This is very useful in functional programs (for example, when passing a random number generator down to recursive calls), but very little work has been done on statistically robust implementations of split ([1,4] are the only examples we know of).

The Random library provides one instance of RandomGen, the abstract data type StdGen:

```
data StdGen = ...      -- Abstract

instance RandomGen StdGen where ...
instance Read      StdGen where ...
instance Show      StdGen where ...

mkStdGen :: Int -> StdGen
```

The result of repeatedly using next should be at least as statistically robust as the "Minimal Standard Random Number Generator" described by [2,3]. Until more is known about implementations of split, all we require is that split deliver generators that are (a) not identical and (b) independently robust in the sense just given.

The show/Read instances of StdGen provide a primitive way to save the state of a random number generator. It is required that read (show g) == g.

In addition, `read` may be used to map an arbitrary string (not necessarily one produced by `show`) onto a value of type `StdGen`. In general, the `read` instance of `StdGen` has the following properties:

- It guarantees to succeed on any string.
- It guarantees to consume only a finite portion of the string.
- Different argument strings are likely to result in different results.

The function `mkStdGen` provides an alternative way of producing an initial generator, by mapping an `Int` into a generator. Again, distinct arguments should be likely to produce distinct generators.

Programmers may, of course, supply their own instances of `RandomGen`.

17.2 The `Random` class

With a source of random number supply in hand, the `Random` class allows the programmer to extract random values of a variety of types:

```
class Random a where
   randomR :: RandomGen g => (a, a) -> g -> (a, g)
   random  :: RandomGen g => g -> (a, g)

   randomRs :: RandomGen g => (a, a) -> g -> [a]
   randoms  :: RandomGen g => g -> [a]

   randomRIO :: (a,a) -> IO a
   randomIO :: IO a

     -- Default methods
   randoms g = x : randoms g'
                   where
                     (x,g') = random g
   randomRs = ...similar...

   randomIO        = getStdRandom random
   randomRIO range = getStdRandom (randomR range)

instance Random Int     where ...
instance Random Integer where ...
instance Random Float   where ...
instance Random Double  where ...
instance Random Bool    where ...
instance Random Char    where ...
```

- `randomR` takes a range (lo, hi) and a random number generator g, and returns a random value uniformly distributed in the closed interval $[lo, hi]$, together with a new generator. It is unspecified what happens if $lo > hi$. For continuous types there is no requirement that the values lo and hi are ever produced, but they may be, depending on the implementation and the interval.
- `random` does the same as `randomR`, but does not take a range.
  - For bounded types (instances of `Bounded`, such as `Char`), the range is normally the whole type.
  - For fractional types, the range is normally the semi-closed interval $[0, 1)$.
  - For `Integer`, the range is (arbitrarily) the range of `Int`.
- The plural versions, `randomRs` and `randoms`, produce an infinite list of random values, and do not return a new generator.
- The `IO` versions, `randomRIO` and `randomIO`, use the global random number generator (see Section 17.3).

17.3 The global random number generator

There is a single, implicit, global random number generator of type `StdGen`, held in some global variable maintained by the `IO` monad. It is initialised automatically in some system-dependent fashion, for example, by using the time of day, or Linux's kernal random number generator. To get deterministic behaviour, use `setStdGen`.

```
setStdGen    :: StdGen -> IO ()
getStdGen    :: IO StdGen
newStdGen    :: IO StdGen
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

- `getStdGen` and `setStdGen` get and set the global random number generator, respectively.
- `newStdGen` applies `split` to the current global random generator, updates it with one of the results, and returns the other.
- `getStdRandom` uses the supplied function to get a value from the current global random generator, and updates the global generator with the new generator returned by the function. For example, `rollDice` gets a random integer between 1 and 6:

  ```
  rollDice :: IO Int
  rollDice = getStdRandom (randomR (1,6))
  ```

References

[1] FW Burton and RL Page, "Distributed random number generation", Journal of Functional Programming, 2(2):203-212, April 1992.

[2] SK Park, and KW Miller, "Random number generators - good ones are hard to find", Comm ACM 31(10), Oct 1988, pp1192-1201.

[3] DG Carta, "Two fast implementations of the minimal standard random number generator", Comm ACM, 33(1), Jan 1990, pp87-88.

[4] P Hellekalek, "Don't trust parallel Monte Carlo", Department of Mathematics, University of Salzburg, `http://random.mat.sbg.ac.at/~peter/pads98.ps`, 1998.

The Web site `http://random.mat.sbg.ac.at/` is a great source of information.

Index

Code entities are shown in `typewriter` font. Ordinary index entries are shown in a roman font.