8 Standard Prelude

In this chapter the entire Haskell Prelude is given. It constitutes a *specification* for the Prelude. Many of the definitions are written with clarity rather than efficiency in mind, and it is not required that the specification be implemented as shown here.

The default method definitions, given with `class` declarations, constitute a specification *only* of the default method. They do not constitute a specification of the meaning of the method in all instances. To take one particular example, the default method for `enumFrom` in class `Enum` will not work properly for types whose range exceeds that of `Int` (because `fromEnum` cannot map all values in the type to distinct `Int` values).

The Prelude shown here is organized into a root module, `Prelude`, and three sub-modules, `PreludeList`, `PreludeText`, and `PreludeIO`. This structure is purely presentational. An implementation is not required to use this organisation for the Prelude, nor are these three modules available for import separately. Only the exports of module `Prelude` are significant.

Some of these modules import Library modules, such as `Char`, `Monad`, `IO`, and `Numeric`. These modules are described fully in Part II. These imports are not, of course, part of the specification of the `Prelude`. That is, an implementation is free to import more, or less, of the Library modules, as it pleases.

Primitives that are not definable in Haskell , indicated by names starting with "`prim`", are defined in a system dependent manner in module `PreludeBuiltin` and are not shown here. Instance declarations that simply bind primitives to class methods are omitted. Some of the more verbose instances with obvious functionality have been left out for the sake of brevity.

Declarations for special types such as `Integer`, or `()` are included in the Prelude for completeness even though the declaration may be incomplete or syntactically invalid. An ellipsis "`...`" is often used in places where the remainder of a definition cannot be given in Haskell.

To reduce the occurrence of unexpected ambiguity errors, and to improve efficiency, a number of commonly-used functions over lists use the `Int` type rather than using a more general numeric type, such as `Integral a` or `Num a`. These functions are: `take`, `drop`, `!!`, `length`, `splitAt`, and `replicate`. The more general versions are given in the `List` library, with the prefix "`generic`"; for example `genericLength`.

```
module Prelude (
    module PreludeList, module PreludeText, module PreludeIO,
    Bool(False, True),
    Maybe(Nothing, Just),
    Either(Left, Right),
    Ordering(LT, EQ, GT),
    Char, String, Int, Integer, Float, Double, Rational, IO,

--      These built-in types are defined in the Prelude, but
--      are denoted by built-in syntax, and cannot legally
--      appear in an export list.
--  List type: []((:), [])
--  Tuple types: (,)((,)), (,,)((,,)), etc.
--  Trivial type: ()(())
--  Functions: (->)

    Eq((==), (/=)),
    Ord(compare, (<), (<=), (>=), (>), max, min),
    Enum(succ, pred, toEnum, fromEnum, enumFrom, enumFromThen,
         enumFromTo, enumFromThenTo),
    Bounded(minBound, maxBound),
    Num((+), (-), (*), negate, abs, signum, fromInteger),
    Real(toRational),
```

```
    Integral(quot, rem, div, mod, quotRem, divMod, toInteger),
    Fractional((/), recip, fromRational),
    Floating(pi, exp, log, sqrt, (**), logBase, sin, cos, tan,
             asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh),
    RealFrac(properFraction, truncate, round, ceiling, floor),
    RealFloat(floatRadix, floatDigits, floatRange, decodeFloat,
              encodeFloat, exponent, significand, scaleFloat, isNaN,
              isInfinite, isDenormalized, isIEEE, isNegativeZero, atan2),
    Monad((>>=), (>>), return, fail),
    Functor(fmap),
    mapM, mapM_, sequence, sequence_, (=<<),
    maybe, either,
    (&&), (||), not, otherwise,
    subtract, even, odd, gcd, lcm, (^), (^^),
    fromIntegral, realToFrac,
    fst, snd, curry, uncurry, id, const, (.), flip, ($), until,
    asTypeOf, error, undefined,
    seq, ($!)
  ) where

import PreludeBuiltin                      -- Contains all `prim' values
import UnicodePrims( primUnicodeMaxChar )  -- Unicode primitives
import PreludeList
import PreludeText
import PreludeIO
import Ratio( Rational )

infixr 9  .
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`
infixl 6  +, -

-- The (:) operator is built-in syntax, and cannot legally be given
-- a fixity declaration; but its fixity is given by:
--   infixr 5  :

infix  4  ==, /=, <, <=, >=, >
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  =<<
infixr 0  $, $!, `seq`

-- Standard types, classes, instances and related functions

-- Equality and Ordered classes


class  Eq a  where
    (==), (/=) :: a -> a -> Bool

        -- Minimal complete definition:
        --      (==) or (/=)
    x /= y     =  not (x == y)
    x == y     =  not (x /= y)


class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a

        -- Minimal complete definition:
        --      (<=) or compare
        -- Using compare can be more efficient for complex types.
    compare x y
         | x == y    =  EQ
         | x <= y    =  LT
         | otherwise =  GT
```

```
    x <= y           =  compare x y /= GT
    x <  y           =  compare x y == LT
    x >= y           =  compare x y /= LT
    x >  y           =  compare x y == GT

-- note that (min x y, max x y) = (x,y) or (y,x)
    max x y
         | x <= y    =  y
         | otherwise =  x
    min x y
         | x <= y    =  x
         | otherwise =  y

-- Enumeration and Bounded classes


class  Enum a  where
    succ, pred       :: a -> a
    toEnum           :: Int -> a
    fromEnum         :: a -> Int
    enumFrom         :: a -> [a]             -- [n..]
    enumFromThen     :: a -> a -> [a]        -- [n,n'..]
    enumFromTo       :: a -> a -> [a]        -- [n..m]
    enumFromThenTo   :: a -> a -> a -> [a]   -- [n,n'..m]

        -- Minimal complete definition:
        --      toEnum, fromEnum
--
-- NOTE: these default methods only make sense for types
--   that map injectively into Int using fromEnum
--  and toEnum.
    succ             =  toEnum . (+1) . fromEnum
    pred             =  toEnum . (subtract 1) . fromEnum
    enumFrom x       =  map toEnum [fromEnum x ..]
    enumFromTo x y   =  map toEnum [fromEnum x .. fromEnum y]
    enumFromThen x y =  map toEnum [fromEnum x, fromEnum y ..]
    enumFromThenTo x y z =
                        map toEnum [fromEnum x, fromEnum y .. fromEnum z]


class  Bounded a  where
    minBound         :: a
    maxBound         :: a

-- Numeric classes


class  (Eq a, Show a) => Num a  where
    (+), (-), (*)    :: a -> a -> a
    negate           :: a -> a
    abs, signum      :: a -> a
    fromInteger      :: Integer -> a

        -- Minimal complete definition:
        --      All, except negate or (-)
    x - y            =  x + negate y
    negate x         =  0 - x


class  (Num a, Ord a) => Real a  where
    toRational       ::  a -> Rational


class  (Real a, Enum a) => Integral a  where
    quot, rem        :: a -> a -> a
    div, mod         :: a -> a -> a
    quotRem, divMod  :: a -> a -> (a,a)
    toInteger        :: a -> Integer
```

```
        -- Minimal complete definition:
        --      quotRem, toInteger
    n `quot` d       =  q  where (q,r) = quotRem n d
    n `rem` d        =  r  where (q,r) = quotRem n d
    n `div` d        =  q  where (q,r) = divMod n d
    n `mod` d        =  r  where (q,r) = divMod n d
    divMod n d       =  if signum r == - signum d then (q-1, r+d) else qr
                        where qr@(q,r) = quotRem n d


class  (Num a) => Fractional a  where
    (/)              :: a -> a -> a
    recip            :: a -> a
    fromRational     :: Rational -> a

        -- Minimal complete definition:
        --      fromRational and (recip or (/))
    recip x          =  1 / x
    x / y            =  x * recip y


class  (Fractional a) => Floating a  where
    pi                  :: a
    exp, log, sqrt      :: a -> a
    (**), logBase       :: a -> a -> a
    sin, cos, tan       :: a -> a
    asin, acos, atan    :: a -> a
    sinh, cosh, tanh    :: a -> a
    asinh, acosh, atanh :: a -> a

        -- Minimal complete definition:
        --      pi, exp, log, sin, cos, sinh, cosh
        --      asin, acos, atan
        --      asinh, acosh, atanh
    x ** y           =  exp (log x * y)
    logBase x y      =  log y / log x
    sqrt x           =  x ** 0.5
    tan  x           =  sin  x / cos  x
    tanh x           =  sinh x / cosh x



class  (Real a, Fractional a) => RealFrac a  where
    properFraction   :: (Integral b) => a -> (b,a)
    truncate, round  :: (Integral b) => a -> b
    ceiling, floor   :: (Integral b) => a -> b

        -- Minimal complete definition:
        --      properFraction
    truncate x       =  m  where (m,_) = properFraction x

    round x          =  let (n,r) = properFraction x
                            m     = if r < 0 then n - 1 else n + 1
                          in case signum (abs r - 0.5) of
                                -1 -> n
                                0  -> if even n then n else m
                                1  -> m

    ceiling x        =  if r > 0 then n + 1 else n
                        where (n,r) = properFraction x

    floor x          =  if r < 0 then n - 1 else n
                        where (n,r) = properFraction x


class  (RealFrac a, Floating a) => RealFloat a  where
    floatRadix       :: a -> Integer
    floatDigits      :: a -> Int
```

```
    floatRange       :: a -> (Int,Int)
    decodeFloat      :: a -> (Integer,Int)
    encodeFloat      :: Integer -> Int -> a
    exponent         :: a -> Int
    significand      :: a -> a
    scaleFloat       :: Int -> a -> a
    isNaN, isInfinite, isDenormalized, isNegativeZero, isIEEE
                     :: a -> Bool
    atan2            :: a -> a -> a

        -- Minimal complete definition:
        --      All except exponent, significand,
        --                 scaleFloat, atan2
    exponent x       =  if m == 0 then 0 else n + floatDigits x
                        where (m,n) = decodeFloat x

    significand x    =  encodeFloat m (- floatDigits x)
                        where (m,_) = decodeFloat x

    scaleFloat k x   =  encodeFloat m (n+k)
                        where (m,n) = decodeFloat x

    atan2 y x
      | x>0           =  atan (y/x)
      | x==0 && y>0   =  pi/2
      | x<0  && y>0   =  pi + atan (y/x)
      |(x<=0 && y<0)  ||
       (x<0 && isNegativeZero y) ||
       (isNegativeZero x && isNegativeZero y)
                      = -atan2 (-y) x
      | y==0 && (x<0 || isNegativeZero x)
                      =  pi    -- must be after the previous test on zero y
      | x==0 && y==0  =  y     -- must be after the other double zero tests
      | otherwise     =  x + y -- x or y is a NaN, return a NaN (via +)

-- Numeric functions


subtract         :: (Num a) => a -> a -> a
subtract         =  flip (-)


even, odd        :: (Integral a) => a -> Bool
even n           =  n `rem` 2 == 0
odd              =  not . even


gcd              :: (Integral a) => a -> a -> a
gcd 0 0          =  error "Prelude.gcd: gcd 0 0 is undefined"
gcd x y          =  gcd' (abs x) (abs y)
                    where gcd' x 0  =  x
                          gcd' x y  =  gcd' y (x `rem` y)


lcm              :: (Integral a) => a -> a -> a
lcm _ 0          =  0
lcm 0 _          =  0
lcm x y          =  abs ((x `quot` (gcd x y)) * y)


(^)              :: (Num a, Integral b) => a -> b -> a
x ^ 0            =  1
x ^ n | n > 0    =  f x (n-1) x
                    where f _ 0 y = y
                          f x n y = g x n  where
                                    g x n | even n  = g (x*x) (n `quot` 2)
                                          | otherwise = f x (n-1) (x*y)
_ ^ _            = error "Prelude.^: negative exponent"
```

```
(^^)             :: (Fractional a, Integral b) => a -> b -> a
x ^^ n           =  if n >= 0 then x^n else recip (x^(-n))


fromIntegral     :: (Integral a, Num b) => a -> b
fromIntegral     =  fromInteger . toInteger


realToFrac     :: (Real a, Fractional b) => a -> b
realToFrac      =  fromRational . toRational

-- Monadic classes


class  Functor f  where
    fmap              :: (a -> b) -> f a -> f b


class  Monad m  where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a

        -- Minimal complete definition:
        --      (>>=), return
    m >> k  =  m >>= \_ -> k
    fail s  = error s


sequence       :: Monad m => [m a] -> m [a]
sequence       =  foldr mcons (return [])
                    where mcons p q = p >>= \x -> q >>= \y -> return (x:y)


sequence_      :: Monad m => [m a] -> m ()
sequence_      =  foldr (>>) (return ())

-- The xxxM functions take list arguments, but lift the function or
-- list element to a monad type

mapM             :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f as        =  sequence (map f as)


mapM_            :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f as       =  sequence_ (map f as)


(=<<)            :: Monad m => (a -> m b) -> m a -> m b
f =<< x          =  x >>= f


-- Trivial type


data  ()  =  ()  deriving (Eq, Ord, Enum, Bounded)
-- Not legal Haskell; for illustration only

-- Function type

-- identity function

id               :: a -> a
id x             =  x

-- constant function
```

```
const            :: a -> b -> a
const x _        =  x

-- function composition

(.)              :: (b -> c) -> (a -> b) -> a -> c
f . g            =  \ x -> f (g x)

-- flip f  takes its (first) two arguments in the reverse order of f.

flip             :: (a -> b -> c) -> b -> a -> c
flip f x y       =  f y x


seq :: a -> b -> b
seq = ...       -- Primitive

-- right-associating infix application operators
-- (useful in continuation-passing style)

($), ($!) :: (a -> b) -> a -> b
f $  x    =  f x
f $! x    =  x `seq` f x


-- Boolean type


data  Bool  =  False | True     deriving (Eq, Ord, Enum, Read, Show, Bounded)

-- Boolean functions


(&&), (||)       :: Bool -> Bool -> Bool
True  && x       =  x
False && _       =  False
True  || _       =  True
False || x       =  x


not              :: Bool -> Bool
not True         =  False
not False        =  True


otherwise        :: Bool
otherwise        =  True


-- Character type


data Char = ... 'a' | 'b' ... -- Unicode values


instance  Eq Char  where
    c == c'          =  fromEnum c == fromEnum c'


instance  Ord Char  where
    c <= c'          =  fromEnum c <= fromEnum c'


instance  Enum Char  where
    toEnum            = primIntToChar
    fromEnum          = primCharToInt
    enumFrom c        = map toEnum [fromEnum c .. fromEnum (maxBound::Char)]
    enumFromThen c c' = map toEnum [fromEnum c, fromEnum c' .. fromEnum lastChar]
                      where lastChar :: Char
```

```
                              lastChar | c' < c    = minBound
                                       | otherwise = maxBound


instance  Bounded Char  where
    minBound  =  '\0'
    maxBound  =  primUnicodeMaxChar


type  String = [Char]


-- Maybe type


data  Maybe a  =  Nothing | Just a      deriving (Eq, Ord, Read, Show)


maybe              :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  =  n
maybe n f (Just x) =  f x


instance  Functor Maybe  where
    fmap f Nothing    =  Nothing
    fmap f (Just x)   =  Just (f x)


instance  Monad Maybe  where
    (Just x) >>= k   =  k x
    Nothing  >>= k   =  Nothing
    return           =  Just
    fail s           =  Nothing

-- Either type


data  Either a b  =  Left a | Right b   deriving (Eq, Ord, Read, Show)


either               :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)  =  f x
either f g (Right y) =  g y

-- IO type


data IO a = ...  -- abstract


instance  Functor IO where
   fmap f x           =  x >>= (return . f)


instance Monad IO where
   (>>=)  = ...
   return = ...
   fail s = ioError (userError s)

-- Ordering type


data  Ordering  =  LT | EQ | GT
          deriving (Eq, Ord, Enum, Read, Show, Bounded)


-- Standard numeric types.  The data declarations for these types cannot
-- be expressed directly in Haskell since the constructor lists would be
-- far too large.
```

```
data  Int  =  minBound ... -1 | 0 | 1 ... maxBound
instance  Eq       Int  where ...
instance  Ord      Int  where ...
instance  Num      Int  where ...
instance  Real     Int  where ...
instance  Integral Int  where ...
instance  Enum     Int  where ...
instance  Bounded  Int  where ...

data  Integer  =  ... -1 | 0 | 1 ...
instance  Eq       Integer  where ...
instance  Ord      Integer  where ...
instance  Num      Integer  where ...
instance  Real     Integer  where ...
instance  Integral Integer  where ...
instance  Enum     Integer  where ...

data  Float
instance  Eq         Float  where ...
instance  Ord        Float  where ...
instance  Num        Float  where ...
instance  Real       Float  where ...
instance  Fractional Float  where ...
instance  Floating   Float  where ...
instance  RealFrac   Float  where ...
instance  RealFloat  Float  where ...

data  Double
instance  Eq         Double  where ...
instance  Ord        Double  where ...
instance  Num        Double  where ...
instance  Real       Double  where ...
instance  Fractional Double  where ...
instance  Floating   Double  where ...
instance  RealFrac   Double  where ...
```

```
instance  RealFloat  Double  where ...

-- The Enum instances for Floats and Doubles are slightly unusual.
-- The `toEnum' function truncates numbers to Int.  The definitions
-- of enumFrom and enumFromThen allow floats to be used in arithmetic
-- series: [0,0.1 .. 0.95].  However, roundoff errors make these somewhat
-- dubious.  This example may have either 10 or 11 elements, depending on
-- how 0.1 is represented.


instance  Enum Float  where
    succ x           =  x+1
    pred x           =  x-1
    toEnum           =  fromIntegral
    fromEnum         =  fromInteger . truncate   -- may overflow
    enumFrom         =  numericEnumFrom
    enumFromThen     =  numericEnumFromThen
    enumFromTo       =  numericEnumFromTo
    enumFromThenTo   =  numericEnumFromThenTo


instance  Enum Double  where
    succ x           =  x+1
    pred x           =  x-1
    toEnum           =  fromIntegral
    fromEnum         =  fromInteger . truncate   -- may overflow
    enumFrom         =  numericEnumFrom
    enumFromThen     =  numericEnumFromThen
    enumFromTo       =  numericEnumFromTo
    enumFromThenTo   =  numericEnumFromThenTo


numericEnumFrom         :: (Fractional a) => a -> [a]

numericEnumFromThen     :: (Fractional a) => a -> a -> [a]

numericEnumFromTo       :: (Fractional a, Ord a) => a -> a -> [a]

numericEnumFromThenTo   :: (Fractional a, Ord a) => a -> a -> a -> [a]
numericEnumFrom         =  iterate (+1)
numericEnumFromThen n m =  iterate (+(m-n)) n
numericEnumFromTo n m   =  takeWhile (<= m+1/2) (numericEnumFrom n)
numericEnumFromThenTo n n' m = takeWhile p (numericEnumFromThen n n')
                             where
                               p | n' >= n   = (<= m + (n'-n)/2)
                                 | otherwise = (>= m + (n'-n)/2)

-- Lists


data  [a]  =  [] | a : [a]  deriving (Eq, Ord)
-- Not legal Haskell; for illustration only


instance Functor [] where
    fmap = map


instance  Monad []  where
    m >>= k          = concat (map k m)
    return x         = [x]
    fail s           = []

-- Tuples


data  (a,b)   =  (a,b)    deriving (Eq, Ord, Bounded)

data  (a,b,c) =  (a,b,c)  deriving (Eq, Ord, Bounded)
```

```
-- Not legal Haskell; for illustration only

-- component projections for pairs:
-- (NB: not provided for triples, quadruples, etc.)

fst              :: (a,b) -> a
fst (x,y)        =  x


snd              :: (a,b) -> b
snd (x,y)        =  y

-- curry converts an uncurried function to a curried function;
-- uncurry converts a curried function to a function on pairs.

curry            :: ((a, b) -> c) -> a -> b -> c
curry f x y      =  f (x, y)


uncurry          :: (a -> b -> c) -> ((a, b) -> c)
uncurry f p      =  f (fst p) (snd p)

-- Misc functions

-- until p f  yields the result of applying f until p holds.

until            :: (a -> Bool) -> (a -> a) -> a -> a
until p f x
     | p x       =  x
     | otherwise =  until p f (f x)

-- asTypeOf is a type-restricted version of const.  It is usually used
-- as an infix operator, and its typing forces its first argument
-- (which is usually overloaded) to have the same type as the second.

asTypeOf         :: a -> a -> a
asTypeOf         =  const

-- error stops execution and displays an error message


error            :: String -> a
error            =  primError

-- It is expected that compilers will recognize this and insert error
-- messages that are more appropriate to the context in which undefined
-- appears.


undefined        :: a
undefined        =  error "Prelude.undefined"
```

8.1 Prelude `PreludeList`

```
-- Standard list functions

module PreludeList (
    map, (++), filter, concat, concatMap,
    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,
    zip, zip3, zipWith, zipWith3, unzip, unzip3)
```

```
  where

import qualified Char(isSpace)

infixl 9  !!
infixr 5  ++
infix  4  `elem`, `notElem`

-- Map and append

map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs


(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)


filter :: (a -> Bool) -> [a] -> [a]
filter p []                 = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs


concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss


concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f

-- head and tail extract the first element and remaining elements,
-- respectively, of a list, which must be non-empty.  last and init
-- are the dual functions working from the end of a finite list,
-- rather than the beginning.


head             :: [a] -> a
head (x:_)       =  x
head []          =  error "Prelude.head: empty list"


tail             :: [a] -> [a]
tail (_:xs)      =  xs
tail []          =  error "Prelude.tail: empty list"


last             :: [a] -> a
last [x]         =  x
last (_:xs)      =  last xs
last []          =  error "Prelude.last: empty list"


init             :: [a] -> [a]
init [x]         =  []
init (x:xs)      =  x : init xs
init []          =  error "Prelude.init: empty list"


null             :: [a] -> Bool
null []          =  True
null (_:_)       =  False

-- length returns the length of a finite list as an Int.

length           :: [a] -> Int
length []        =  0
```

```
length (_:l)     =  1 + length l

-- List index (subscript) operator, 0-origin

(!!)                :: [a] -> Int -> a
xs     !! n | n < 0 =  error "Prelude.!!: negative index"
[]     !! _         =  error "Prelude.!!: index too large"
(x:_)  !! 0         =  x
(_:xs) !! n         =  xs !! (n-1)

-- foldl, applied to a binary operator, a starting value (typically the
-- left-identity of the operator), and a list, reduces the list using
-- the binary operator, from left to right:
--  foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...) `f` xn
-- foldl1 is a variant that has no starting value argument, and  thus must
-- be applied to non-empty lists.  scanl is similar to foldl, but returns
-- a list of successive reduced values from the left:
--      scanl f z [x1, x2, ...] == [z, z `f` x1, (z `f` x1) `f` x2, ...]
-- Note that  last (scanl f z xs) == foldl f z xs.
-- scanl1 is similar, again without the starting element:
--      scanl1 f [x1, x2, ...] == [x1, x1 `f` x2, ...]


foldl            :: (a -> b -> a) -> a -> [b] -> a
foldl f z []     =  z
foldl f z (x:xs) =  foldl f (f z x) xs


foldl1           :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)  =  foldl f x xs
foldl1 _ []      =  error "Prelude.foldl1: empty list"


scanl            :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q xs     =  q : (case xs of
                            []   -> []
                            x:xs -> scanl f (f q x) xs)


scanl1           :: (a -> a -> a) -> [a] -> [a]
scanl1 f (x:xs)  =  scanl f x xs
scanl1 _ []      =  []

-- foldr, foldr1, scanr, and scanr1 are the right-to-left duals of the
-- above functions.


foldr            :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)


foldr1           :: (a -> a -> a) -> [a] -> a
foldr1 f [x]     =  x
foldr1 f (x:xs)  =  f x (foldr1 f xs)
foldr1 _ []      =  error "Prelude.foldr1: empty list"


scanr             :: (a -> b -> b) -> b -> [a] -> [b]
scanr f q0 []     =  [q0]
scanr f q0 (x:xs) =  f x q : qs
                     where qs@(q:_) = scanr f q0 xs


scanr1          :: (a -> a -> a) -> [a] -> [a]
scanr1 f []     =  []
scanr1 f [x]    =  [x]
scanr1 f (x:xs) =  f x q : qs
                   where qs@(q:_) = scanr1 f xs
```

```
-- iterate f x returns an infinite list of repeated applications of f to x:
-- iterate f x == [x, f x, f (f x), ...]

iterate          :: (a -> a) -> a -> [a]
iterate f x      =  x : iterate f (f x)

-- repeat x is an infinite list, with x the value of every element.

repeat           :: a -> [a]
repeat x         =  xs where xs = x:xs

-- replicate n x is a list of length n with x the value of every element

replicate        :: Int -> a -> [a]
replicate n x    =  take n (repeat x)

-- cycle ties a finite list into a circular one, or equivalently,
-- the infinite repetition of the original list.  It is the identity
-- on infinite lists.


cycle            :: [a] -> [a]
cycle []         =  error "Prelude.cycle: empty list"
cycle xs         =  xs' where xs' = xs ++ xs'

-- take n, applied to a list xs, returns the prefix of xs of length n,
-- or xs itself if n > length xs.  drop n xs returns the suffix of xs
-- after the first n elements, or [] if n > length xs.  splitAt n xs
-- is equivalent to (take n xs, drop n xs).


take                   :: Int -> [a] -> [a]
take n _      | n <= 0 =  []
take _ []              =  []
take n (x:xs)          =  x : take (n-1) xs


drop                   :: Int -> [a] -> [a]
drop n xs     | n <= 0 =  xs
drop _ []              =  []
drop n (_:xs)          =  drop (n-1) xs


splitAt                  :: Int -> [a] -> ([a],[a])
splitAt n xs             =  (take n xs, drop n xs)

-- takeWhile, applied to a predicate p and a list xs, returns the longest
-- prefix (possibly empty) of xs of elements that satisfy p.  dropWhile p xs
-- returns the remaining suffix.  span p xs is equivalent to
-- (takeWhile p xs, dropWhile p xs), while break p uses the negation of p.


takeWhile               :: (a -> Bool) -> [a] -> [a]
takeWhile p []          =  []
takeWhile p (x:xs)
            | p x       =  x : takeWhile p xs
            | otherwise =  []


dropWhile               :: (a -> Bool) -> [a] -> [a]
dropWhile p []          =  []
dropWhile p xs@(x:xs')
            | p x       =  dropWhile p xs'
            | otherwise =  xs


span, break             :: (a -> Bool) -> [a] -> ([a],[a])
span p []            = ([],[])
```

```
span p xs@(x:xs')
            | p x       =  (x:ys,zs)
            | otherwise =  ([],xs)
                           where (ys,zs) = span p xs'

break p                 =  span (not . p)

-- lines breaks a string up into a list of strings at newline characters.
-- The resulting strings do not contain newlines.  Similary, words
-- breaks a string up into a list of words, which were delimited by
-- white space.  unlines and unwords are the inverse operations.
-- unlines joins lines with terminating newlines, and unwords joins
-- words with separating spaces.


lines            :: String -> [String]
lines ""         =  []
lines s          =  let (l, s') = break (== '\n') s
                      in  l : case s' of
                                []      -> []
                                (_:s'') -> lines s''


words            :: String -> [String]
words s          =  case dropWhile Char.isSpace s of
                      "" -> []
                      s' -> w : words s''
                            where (w, s'') = break Char.isSpace s'


unlines          :: [String] -> String
unlines          =  concatMap (++ "\n")


unwords          :: [String] -> String
unwords []       =  ""
unwords ws       =  foldr1 (\w s -> w ++ ' ':s) ws

-- reverse xs returns the elements of xs in reverse order.  xs must be finite.

reverse          :: [a] -> [a]
reverse          =  foldl (flip (:)) []

-- and returns the conjunction of a Boolean list.  For the result to be
-- True, the list must be finite; False, however, results from a False
-- value at a finite index of a finite or infinite list.  or is the
-- disjunctive dual of and.

and, or          :: [Bool] -> Bool
and              =  foldr (&&) True
or               =  foldr (||) False

-- Applied to a predicate and a list, any determines if any element
-- of the list satisfies the predicate.  Similarly, for all.

any, all         :: (a -> Bool) -> [a] -> Bool
any p            =  or . map p
all p            =  and . map p

-- elem is the list membership predicate, usually written in infix form,
-- e.g., x `elem` xs.  notElem is the negation.

elem, notElem    :: (Eq a) => a -> [a] -> Bool
elem x           =  any (== x)
notElem x        =  all (/= x)

-- lookup key assocs looks up a key in an association list.

lookup           :: (Eq a) => a -> [(a,b)] -> Maybe b
```

```
lookup key []    =  Nothing
lookup key ((x,y):xys)
     | key == x   =  Just y
     | otherwise  =  lookup key xys

-- sum and product compute the sum or product of a finite list of numbers.

sum, product     :: (Num a) => [a] -> a
sum              =  foldl (+) 0
product          =  foldl (*) 1

-- maximum and minimum return the maximum or minimum value from a list,
-- which must be non-empty, finite, and of an ordered type.

maximum, minimum :: (Ord a) => [a] -> a
maximum []       =  error "Prelude.maximum: empty list"
maximum xs       =  foldl1 max xs

minimum []       =  error "Prelude.minimum: empty list"
minimum xs       =  foldl1 min xs

-- zip takes two lists and returns a list of corresponding pairs.  If one
-- input list is short, excess elements of the longer list are discarded.
-- zip3 takes three lists and returns a list of triples.  Zips for larger
-- tuples are in the List library


zip              :: [a] -> [b] -> [(a,b)]
zip              =  zipWith (,)


zip3             :: [a] -> [b] -> [c] -> [(a,b,c)]
zip3             =  zipWith3 (,,)

-- The zipWith family generalises the zip family by zipping with the
-- function given as the first argument, instead of a tupling function.
-- For example, zipWith (+) is applied to two lists to produce the list
-- of corresponding sums.


zipWith          :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)
                 =  z a b : zipWith z as bs
zipWith _ _ _    =  []


zipWith3         :: (a->b->c->d) -> [a]->[b]->[c]->[d]
zipWith3 z (a:as) (b:bs) (c:cs)
                 =  z a b c : zipWith3 z as bs cs
zipWith3 _ _ _ _ =  []


-- unzip transforms a list of pairs into a pair of lists.


unzip            :: [(a,b)] -> ([a],[b])
unzip            =  foldr (\(a,b) ~(as,bs) -> (a:as,b:bs)) ([],[])


unzip3           :: [(a,b,c)] -> ([a],[b],[c])
unzip3           =  foldr (\(a,b,c) ~(as,bs,cs) -> (a:as,b:bs,c:cs))
                          ([],[],[])
```

8.2 Prelude `PreludeText`

```
module PreludeText (
    ReadS, ShowS,
    Read(readsPrec, readList),
    Show(showsPrec, show, showList),
    reads, shows, read, lex,
    showChar, showString, readParen, showParen ) where

-- The instances of Read and Show for
--      Bool, Maybe, Either, Ordering
-- are done via "deriving" clauses in Prelude.hs

import Char(isSpace, isAlpha, isDigit, isAlphaNum,
            showLitChar, readLitChar, lexLitChar)

import Numeric(showSigned, showInt, readSigned, readDec, showFloat,
               readFloat, lexDigits)


type  ReadS a  = String -> [(a,String)]

type  ShowS    = String -> String


class  Read a  where
    readsPrec        :: Int -> ReadS a
    readList         :: ReadS [a]

        -- Minimal complete definition:
        --      readsPrec
    readList         = readParen False (\r -> [pr | ("[",s)  <- lex r,
                                                    pr       <- readl s])
                       where readl  s = [([],t)   | ("]",t)  <- lex s] ++
                                        [(x:xs,u) | (x,t)    <- reads s,
                                                    (xs,u)   <- readl' t]
                             readl' s = [([],t)   | ("]",t)  <- lex s] ++
                                        [(x:xs,v) | (",",t)  <- lex s,
                                                    (x,u)    <- reads t,
                                                    (xs,v)   <- readl' u]


class  Show a  where
    showsPrec        :: Int -> a -> ShowS
    show             :: a -> String
    showList         :: [a] -> ShowS

        -- Mimimal complete definition:
        --      show or showsPrec
    showsPrec _ x s   = show x ++ s

    show x            = showsPrec 0 x ""

    showList []       = showString "[]"
    showList (x:xs)   = showChar '[' . shows x . showl xs
                        where showl []     = showChar ']'
                              showl (x:xs) = showChar ',' . shows x .
                                             showl xs


reads            :: (Read a) => ReadS a
reads            =  readsPrec 0


shows            :: (Show a) => a -> ShowS
shows            =  showsPrec 0


read             :: (Read a) => String -> a
read s           =  case [x | (x,t) <- reads s, ("","") <- lex t] of
```

```
                          [x] -> x
                          []  -> error "Prelude.read: no parse"
                          _   -> error "Prelude.read: ambiguous parse"


showChar          :: Char -> ShowS
showChar          =  (:)


showString        :: String -> ShowS
showString        =  (++)


showParen         :: Bool -> ShowS -> ShowS
showParen b p     =  if b then showChar '(' . p . showChar ')' else p


readParen         :: Bool -> ReadS a -> ReadS a
readParen b g     =  if b then mandatory else optional
                     where optional r  = g r ++ mandatory r
                           mandatory r = [(x,u) | ("(",s) <- lex r,
                                                  (x,t)   <- optional s,
                                                  (")",u) <- lex t    ]

-- This lexer is not completely faithful to the Haskell lexical syntax.
-- Current limitations:
--    Qualified names are not handled properly
--    Octal and hexidecimal numerics are not recognized as a single token
--    Comments are not treated properly


lex               :: ReadS String
lex ""            =  [("","")]
lex (c:s)
   | isSpace c    =  lex (dropWhile isSpace s)
lex ('\'':s)      =  [('\'':ch++"'", t) | (ch,'\'':t)  <- lexLitChar s,
                                          ch /= "'" ]
lex ('"':s)       =  [('"':str, t)      | (str,t) <- lexString s]
                     where
                     lexString ('"':s) = [("\"",s)]
                     lexString s = [(ch++str, u)
                                          | (ch,t)  <- lexStrItem s,
                                            (str,u) <- lexString t  ]

                     lexStrItem ('\\':'&':s) =  [("\\&",s)]
                     lexStrItem ('\\':c:s) | isSpace c
                                           =  [("\\&",t) |
                                               '\\':t <-
                                                   [dropWhile isSpace s]]
                     lexStrItem s          =  lexLitChar s

lex (c:s) | isSingle c = [([c],s)]
          | isSym c    = [(c:sym,t)       | (sym,t) <- [span isSym s]]
          | isAlpha c  = [(c:nam,t)       | (nam,t) <- [span isIdChar s]]
          | isDigit c  = [(c:ds++fe,t)    | (ds,s)  <- [span isDigit s],
                                            (fe,t)  <- lexFracExp s     ]
          | otherwise  = []    -- bad character
             where
              isSingle c =  c `elem` ",;()[]{}_`"
              isSym c    =  c `elem` "!@#$%&*+./<=>?\\^|:-~"
              isIdChar c =  isAlphaNum c || c `elem` "_'"

              lexFracExp ('.':c:cs) | isDigit c
                            = [('.':ds++e,u) | (ds,t) <- lexDigits (c:cs),
                                               (e,u)  <- lexExp t]
              lexFracExp s  = lexExp s

              lexExp (e:s) | e `elem` "eE"
                       = [(e:c:ds,u) | (c:t)  <- [s], c `elem` "+-",
```

```
                                                   (ds,u) <- lexDigits t] ++
                             [(e:ds,t)   | (ds,t) <- lexDigits s]
                  lexExp s = [("",s)]


instance  Show Int  where
    showsPrec n = showsPrec n . toInteger
        -- Converting to Integer avoids
        -- possible difficulty with minInt


instance  Read Int  where
  readsPrec p r = [(fromInteger i, t) | (i,t) <- readsPrec p r]
        -- Reading at the Integer type avoids
        -- possible difficulty with minInt


instance  Show Integer  where
    showsPrec           = showSigned showInt


instance  Read Integer  where
    readsPrec p         = readSigned readDec


instance  Show Float  where
    showsPrec p         = showFloat


instance  Read Float  where
    readsPrec p         = readSigned readFloat


instance  Show Double  where
    showsPrec p         = showFloat


instance  Read Double  where
    readsPrec p         = readSigned readFloat


instance  Show ()  where
    showsPrec p () = showString "()"


instance Read () where
    readsPrec p    = readParen False
                            (\r -> [((),t) | ("(",s) <- lex r,
                                             (")",t) <- lex s ] )

instance  Show Char  where
    showsPrec p '\'' = showString "'\\''"
    showsPrec p c    = showChar '\'' . showLitChar c . showChar '\''

    showList cs = showChar '"' . showl cs
                 where showl ""       = showChar '"'
                       showl ('"':cs) = showString "\\\"" . showl cs
                       showl (c:cs)   = showLitChar c . showl cs


instance  Read Char  where
    readsPrec p      = readParen False
                            (\r -> [(c,t) | ('\'':s,t)<- lex r,
                                            (c,"\'")  <- readLitChar s])

    readList = readParen False (\r -> [(l,t) | ('"':s, t) <- lex r,
                                               (l,_)      <- readl s ])
        where readl ('"':s)      = [("",s)]
              readl ('\\':'&':s) = readl s
```

```
              readl s             = [(c:cs,u) | (c ,t) <- readLitChar s,
                                                (cs,u) <- readl t       ]


instance  (Show a) => Show [a]  where
    showsPrec p      = showList


instance  (Read a) => Read [a]  where
    readsPrec p      = readList

-- Tuples


instance  (Show a, Show b) => Show (a,b)  where
    showsPrec p (x,y) = showChar '(' . shows x . showChar ',' .
                                       shows y . showChar ')'


instance  (Read a, Read b) => Read (a,b)  where
    readsPrec p       = readParen False
                            (\r -> [((x,y), w) | ("(",s) <- lex r,
                                                  (x,t)   <- reads s,
                                                  (",",u) <- lex t,
                                                  (y,v)   <- reads u,
                                                  (")",w) <- lex v ] )

-- Other tuples have similar Read and Show instances
```

8.3 Prelude `PreludeIO`

```
module PreludeIO (
    FilePath, IOError, ioError, userError, catch,
    putChar, putStr, putStrLn, print,
    getChar, getLine, getContents, interact,
    readFile, writeFile, appendFile, readIO, readLn
  ) where

import PreludeBuiltin



type  FilePath = String


data IOError    -- The internals of this type are system dependent


instance  Show IOError  where ...

instance  Eq IOError  where ...


ioError    ::  IOError -> IO a
ioError    =   primIOError


userError  ::  String -> IOError
userError  =   primUserError


catch      ::  IO a -> (IOError -> IO a) -> IO a
catch      =   primCatch
```

```
putChar    :: Char -> IO ()
putChar    =  primPutChar


putStr     :: String -> IO ()
putStr s   =  mapM_ putChar s


putStrLn   :: String -> IO ()
putStrLn s =  do putStr s
                 putStr "\n"


print      :: Show a => a -> IO ()
print x    =  putStrLn (show x)


getChar    :: IO Char
getChar    =  primGetChar


getLine    :: IO String
getLine    =  do c <- getChar
                 if c == '\n' then return "" else
                    do s <- getLine
                       return (c:s)


getContents :: IO String
getContents =  primGetContents


interact    ::  (String -> String) -> IO ()
-- The hSetBuffering ensures the expected interactive behaviour
interact f  =  do hSetBuffering stdin  NoBuffering
                  hSetBuffering stdout NoBuffering
                  s <- getContents
                  putStr (f s)


readFile   :: FilePath -> IO String
readFile   =  primReadFile


writeFile  :: FilePath -> String -> IO ()
writeFile  =  primWriteFile


appendFile :: FilePath -> String -> IO ()
appendFile =  primAppendFile

  -- raises an exception instead of an error

readIO   :: Read a => String -> IO a
readIO s =  case [x | (x,t) <- reads s, ("","") <- lex t] of
              [x] -> return x
              []  -> ioError (userError "Prelude.readIO: no parse")
              _   -> ioError (userError "Prelude.readIO: ambiguous parse")


readLn :: Read a => IO a
readLn =  do l <- getLine
             r <- readIO l
             return r
```

The Haskell 98 Report
top | back | next | contents | function index
December 2002