Appendix B Standard prelude

In this appendix we present some of the most commonly used definitions from the Haskell standard prelude. For expository purposes, a number of the definitions are presented in simplified form. The full version of the prelude is available from the Haskell home page, `http://www.haskell.org`.

B.1 Basic classes

Equality types:

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

   x /= y = not (x == y)
```

Ordered types:

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

   min x y | x <= y    = x
           | otherwise = y

   max x y | x <= y    = y
           | otherwise = x
```

Showable types:

```
class Show a where
   show :: a -> String
```

Readable types:

```
class Read a where
   read :: String -> a
```

Numeric types:

```
class Num a where
   (+), (-), (*)       :: a -> a -> a
   negate, abs, signum :: a -> a
```

Integral types:

```
class Num a => Integral a where
   div, mod :: a -> a -> a
```

Fractional types:

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

   recip n = 1/n
```

B.2 Booleans

Type declaration:

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

Logical conjunction:

```
(&&) :: Bool -> Bool -> Bool
False && _ = False
True  && b = b
```

Logical disjunction:

```
(||) :: Bool -> Bool -> Bool
False || b = b
True  || _ = True
```

Logical negation:

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

Guard that always succeeds:

```
otherwise :: Bool
otherwise = True
```

B.3 Characters

Type declaration:

```
data Char = ...
            deriving (Eq, Ord, Show, Read)
```

The definitions below are provided in the library `Data.Char`, which can be loaded by entering the following in GHCi or at the start of a script:

```
import Data.Char
```

Decide if a character is a lower-case letter:

```
isLower :: Char -> Bool
isLower c = c >= 'a' && c <= 'z'
```

Decide if a character is an upper-case letter:

```
isUpper :: Char -> Bool
isUpper c = c >= 'A' && c <= 'Z'
```

Decide if a character is alphabetic:

```
isAlpha :: Char -> Bool
isAlpha c = isLower c || isUpper c
```

Decide if a character is a digit:

```
isDigit :: Char -> Bool
isDigit c = c >= '0' && c <= '9'
```

Decide if a character is alpha-numeric:

```
isAlphaNum :: Char -> Bool
isAlphaNum c = isAlpha c || isDigit c
```

Decide if a character is spacing:

```
isSpace :: Char -> Bool
isSpace c = elem c " \t\n"
```

Convert a character to a Unicode number:

```
ord :: Char -> Int
ord c = ...
```

Convert a Unicode number to a character:

```
chr :: Int -> Char
chr n = ...
```

Convert a digit to an integer:

```
digitToInt :: Char -> Int
digitToInt c | isDigit c = ord c - ord '0'
```

Convert an integer to a digit:

```
intToDigit :: Int -> Char
intToDigit n | n >= 0 && n <= 9 = chr (ord '0' + n)
```

Convert a letter to lower-case:

```
toLower :: Char -> Char
toLower c | isUpper c = chr (ord c - ord 'A' + ord 'a')
          | otherwise = c
```

Convert a letter to upper-case:

```
toUpper :: Char -> Char
toUpper c | isLower c = chr (ord c - ord 'a' + ord 'A')
          | otherwise = c
```

B.4 Strings

Type declaration:

```
type String = [Char]
```

B.5 Numbers

Type declarations:

```
data Int = ...
           deriving (Eq, Ord, Show, Read, Num, Integral)

data Integer = ...
               deriving (Eq, Ord, Show, Read, Num, Integral)

data Float = ...
             deriving (Eq, Ord, Show, Read, Num, Fractional)

data Double = ...
              deriving (Eq, Ord, Show, Read, Num, Fractional)
```

Decide if an integer is even:

```
even :: Integral a => a -> Bool
even n = n `mod` 2 == 0
```

Decide if an integer is odd:

```
odd :: Integral a => a -> Bool
odd = not . even
```

Exponentiation:

```
(^) :: (Num a, Integral b) => a -> b -> a
_ ^ 0 = 1
x ^ n = x * (x ^ (n-1))
```

B.6 Tuples

Type declarations:

```
data () = ...
          deriving (Eq, Ord, Show, Read)

data (a,b) = ...
             deriving (Eq, Ord, Show, Read)

data (a,b,c) = ...
               deriving (Eq, Ord, Show, Read)
```

Select the first component of a pair:

```
fst :: (a,b) -> a
fst (x,_) = x
```

Select the second component of a pair:

```
snd :: (a,b) -> b
snd (_,y) = y
```

Convert a function on pairs to a curried function:

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

Convert a curried function to a function on pairs:

```
uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f = \(x,y) -> f x y
```

B.7 Maybe

Type declaration:

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

B.8 Lists

Type declaration:

```
data [a] = [] | a:[a]
           deriving (Eq, Ord, Show, Read)
```

Select the first element of a non-empty list:

```
head :: [a] -> a
head (x:_) = x
```

Select the last element of a non-empty list:

```
last :: [a] -> a
last [x]    = x
last (_:xs) = last xs
```

Select the nth element of a non-empty list:

```
(!!) :: [a] -> Int -> a
(x:_)  !! 0 = x
(_:xs) !! n = xs !! (n-1)
```

Select the first n elements of a list:

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

Select all elements of a list that satisfy a predicate:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
```

Select elements of a list while they satisfy a predicate:

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

Remove the first element from a non-empty list:

```
tail :: [a] -> [a]
tail (_:xs) = xs
```

Remove the last element from a non-empty list:

```
init :: [a] -> [a]
init [_]    = []
init (x:xs) = x : init xs
```

Remove the first n elements from a list:

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

Remove elements from a list while they satisfy a predicate:

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

Split a list at the nth element:

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

Produce an infinite list of identical elements:

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

Produce a list with n identical elements:

```
replicate :: Int -> a -> [a]
replicate n = take n . repeat
```

Produce an infinite list by iterating a function over a value:

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

Produce a list of pairs from a pair of lists:

```
zip :: [a] -> [b] -> [(a,b)]
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

Append two lists:

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

Reverse a list:

```
reverse :: [a] -> [a]
reverse = foldl (\xs x -> x:xs) []
```

Apply a function to all elements of a list:

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

B.9 Functions

Type declaration:

```
data a -> b = ...
```

Identity function:

```
id :: a -> a
id = \x -> x
```

Function composition:

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

Constant functions:

```
const :: a -> (b -> a)
const x = \_ -> x
```

Strict application:

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

Flip the arguments of a curried function:

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

B.10 Input/output

Type declaration:

```
data IO a = ...
```

Read a character from the keyboard:

```
getChar :: IO Char
getChar = ...
```

Read a string from the keyboard:

```
getLine :: IO String
getLine = do x <- getChar
             if x == '\n' then
                return ""
             else
                do xs <- getLine
                   return (x:xs)
```

Read a value from the keyboard:

```
readLn :: Read a => IO a
readLn = do xs <- getLine
            return (read xs)
```

Write a character to the screen:

```
putChar :: Char -> IO ()
putChar c = ...
```

Write a string to the screen:

```
putStr :: String -> IO ()
putStr ""     = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

Write a string to the screen and move to a new line:

```
putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

Write a value to the screen:

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

Display an error message and terminate the program:

```
error :: String -> a
error xs = ...
```

B.11 Functors

Class declaration:

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

Maybe functor:

```
instance Functor Maybe where
   -- fmap :: (a -> b) -> Maybe a -> Maybe b
   fmap _ Nothing  = Nothing
   fmap g (Just x) = Just (g x)
```

List functor:

```
instance Functor [] where
   -- fmap :: (a -> b) -> [a] -> [b]
   fmap = map
```

IO functor:

```
instance Functor IO where
   -- fmap :: (a -> b) -> IO a -> IO b
   fmap g mx = do {x <- mx; return (g x)}
```

Infix version of `fmap`:

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
g <$> x = fmap g x
```

B.12 Applicatives

Class declaration:

```
class Functor f => Applicative f where
   pure  :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
```

Maybe applicative:

```
instance Applicative Maybe where
   -- pure :: a -> Maybe a
   pure = Just

   -- (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
   Nothing  <*> _  = Nothing
   (Just g) <*> mx = fmap g mx
```

List applicative:

```
instance Applicative [] where
   -- pure :: a -> [a]
   pure x = [x]

   -- (<*>) :: [a -> b] -> [a] -> [b]
```

```
   gs <*> xs = [g x | g <- gs, x <- xs]
```

IO applicative:

```
instance Applicative IO where
  -- pure :: a -> IO a
  pure = return

  -- (<*>) :: IO (a -> b) -> IO a -> IO b
  mg <*> mx = do {g <- mg; x <- mx; return (g x)}
```

B.13 Monads

Class declaration:

```
class Applicative m => Monad m where
   return :: a -> m a
   (>>=)  :: m a -> (a -> m b) -> m b

   return = pure
```

Maybe monad:

```
instance Monad Maybe where
   -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
   Nothing  >>= _ = Nothing
   (Just x) >>= f = f x
```

List monad:

```
instance Monad [] where
   -- (>>=) :: [a] -> (a -> [b]) -> [b]
   xs >>= f = [y | x <- xs, y <- f x]
```

IO monad:

```
instance Monad IO where
   -- return :: a -> IO a
   return x = ...

   -- (>>=) :: IO a -> (a -> IO b) -> IO b
   mx >>= f = ...
```

B.14 Alternatives

The declarations below are provided in the library `Control.Applicative`, which can be loaded by entering the following in GHCi or at the start of a script:

```
import Control.Applicative
```

Class declaration:

```
class Applicative f => Alternative f where
   empty :: f a
   (<|>) :: f a -> f a -> f a
   many  :: f a -> f [a]
   some  :: f a -> f [a]

   many x = some x <|> pure []
   some x = pure (:) <*> x <*> many x
```

Maybe alternative:

```
instance Alternative Maybe where
   -- empty :: Maybe a
   empty = Nothing

   -- (<|>) :: Maybe a -> Maybe a -> Maybe a
   Nothing  <|> my = my
   (Just x) <|> _  = Just x
```

List alternative:

```
instance Alternative [] where
   -- empty :: [a]
   empty = []

   -- (<|>) :: [a] -> [a] -> [a]
   (<|>) = (++)
```

B.15 MonadPlus

The declarations below are provided in the library `Control.Monad`, which can be loaded by entering the following in GHCi or at the start of a script:

```
import Control.Monad
```

Class declaration:

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

   mzero = empty
   mplus = (<|>)
```

Maybe monadplus:

```
instance MonadPlus Maybe
```

List monadplus:

```
instance MonadPlus []
```

B.16 Monoids

Class declaration:

```
class Monoid a where
   mempty  :: a
   mappend :: a -> a -> a

   mconcat :: [a] -> a
   mconcat = foldr mappend mempty
```

The declarations below are provided in a library `Data.Monoid`, which can be loaded by entering the following in GHCi or at the start of a script:

```
import Data.Monoid
```

Maybe monoid:

```
instance Monoid a => Monoid (Maybe a) where
   -- mempty :: Maybe a
   mempty = Nothing

   -- mappend :: Maybe a -> Maybe a -> Maybe a
   Nothing  `mappend` my       = my
   mx       `mappend` Nothing  = mx
   Just x   `mappend` Just y   = Just (x `mappend` y)
```

List monoid:

```
instance Monoid [a] where
   -- mempty :: [a]
   mempty = []

   -- mappend :: [a] -> [a] -> [a]
   mappend = (++)
```

Numeric monoid for addition:

```
newtype Sum a = Sum a
                deriving (Eq, Ord, Show, Read)
```

```
getSum :: Sum a -> a
getSum (Sum x) = x

instance Num a => Monoid (Sum a) where
   -- mempty :: Sum a
   mempty = Sum 0

   -- mappend :: Sum a -> Sum a -> Sum a
   Sum x `mappend` Sum y = Sum (x+y)
```

Numeric monoid for multiplication:

```
newtype Product a = Product a
                    deriving (Eq, Ord, Show, Read)

getProduct :: Product a -> a
getProduct (Product x) = x

instance Num a => Monoid (Product a) where
   -- mempty :: Product a
   mempty = Product 1

   -- mappend :: Product a -> Product a -> Product a
   Product x `mappend` Product y = Product (x*y)
```

Boolean monoid for conjunction:

```
newtype All = All Bool
              deriving (Eq, Ord, Show, Read)

getAll :: All -> Bool
getAll (All b) = b

instance Monoid All where
   -- mempty :: All
   mempty = All True

   -- mappend :: All -> All -> All
   All b `mappend` All c = All (b && c)
```

Boolean monoid for disjunction:

```
newtype Any = Any Bool
              deriving (Eq, Ord, Show, Read)

getAny :: Any -> Bool
getAny (Any b) = b
```

```
instance Monoid Any where
   -- mempty :: Any
   mempty = Any False

   -- mappend :: Any -> Any -> Any
   Any b `mappend` Any c = Any (b || c)
```

Infix version of `mappend`:

```
(<>) :: Monoid a => a -> a -> a
x <> y = x `mappend` y
```

B.17 Foldables

The declarations below are provided in the library `Data.Foldable`, which can be loaded by entering the following in GHCi or at the start of a script:

```
import Data.Foldable
```

Class declaration:

```
class Foldable t where
   foldMap :: Monoid b => (a -> b) -> t a -> b
   foldr   :: (a -> b -> b) -> b -> t a -> b

   fold    :: Monoid a => t a -> a
   foldl   :: (a -> b -> a) -> a -> t b -> a
   foldr1  :: (a -> a -> a) -> t a -> a
   foldl1  :: (a -> a -> a) -> t a -> a

   toList  :: t a -> [a]
   null    :: t a -> Bool
   length  :: t a -> Int
   elem    :: Eq a => a -> t a -> Bool
   maximum :: Ord a => t a -> a
   minimum :: Ord a => t a -> a
   sum     :: Num a => t a -> a
   product :: Num a => t a -> a
```

Default definitions:

```
   foldMap f = foldr (mappend . f) mempty
   foldr f v = foldr f v . toList

   fold      = foldMap id
```

```
foldl f v = foldl f v . toList
foldr1 f  = foldr1 f . toList
foldl1 f  = foldl1 f . toList

toList    = foldMap (\x -> [x])
null      = null . toList
length    = length . toList
elem x    = elem x . toList
maximum   = maximum . toList
minimum   = minimum . toList
sum       = sum . toList
product   = product . toList
```

The minimal complete definition for an instance is to define `foldMap` or `foldr`, as all other functions in the class can be derived from either of these two using the above default definitions and the following instance for lists.

List foldable:

```
instance Foldable [] where

   -- foldMap :: Monoid b => (a -> b) -> [a] -> b
   foldMap _ []     = mempty
   foldMap f (x:xs) = f x `mappend` foldMap f xs

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

   -- fold :: Monoid a => [a] -> a
   fold = foldMap id

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

   -- foldr1 :: (a -> a -> a) -> [a] -> a
   foldr1 _ [x]    = x
   foldr1 f (x:xs) = f x (foldr1 f xs)

   -- foldl1 :: (a -> a -> a) -> [a] -> a
   foldl1 f (x:xs) = foldl f x xs

   -- toList :: [a] -> [a]
   toList = id
```

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

-- length :: [a] -> Int
length = foldl (\n _ -> n+1) 0

-- elem :: Eq a => a -> [a] -> Bool
elem x xs = any (==x) xs

-- maximum :: Ord a => [a] -> a
maximum = foldl1 max

-- minimum :: Ord a => [a] -> a
minimum = foldl1 min

-- sum :: Num a => [a] -> a
sum = foldl (+) 0

-- product :: Num a => [a] -> a
product = foldl (*) 1
```

Decide if all logical values in a structure are `True`:

```
and :: Foldable t => t Bool -> Bool
and = getAll . foldMap All
```

Decide if any logical value in a structure is `True`:

```
or :: Foldable t => t Bool -> Bool
or = getAny . foldMap Any
```

Decide if all elements in a structure satisfy a predicate:

```
all :: Foldable t => (a -> Bool) -> t a -> Bool
all p = getAll . foldMap (All . p)
```

Decide if any element in a structure satisfies a predicate:

```
any :: Foldable t => (a -> Bool) -> t a -> Bool
any p = getAny . foldMap (Any . p)
```

Concatenate a structure whose elements are lists:

```
concat :: Foldable t => t [a] -> [a]
concat = fold
```

B.18 Traversables

Class declaration:

```
class (Functor t, Foldable t) => Traversable t where
    traverse  :: Applicative f => (a -> f b) -> t a -> f (t b)
    sequenceA :: Applicative f => t (f a) -> f (t a)

    mapM      :: Monad m => (a -> m b) -> t a -> m (t b)
    sequence  :: Monad m => t (m a) -> m (t a)
```

Default definitions:

```
    traverse g = sequenceA . fmap g
    sequenceA  = traverse id

    mapM       = traverse
    sequence   = sequenceA
```

The minimal complete definition for an instance of the class is to define `traverse` or `sequenceA`, as all other functions in the class can be derived from either of these two using the above default definitions.

Maybe traversable:

```
instance Traversable Maybe where
    -- traverse :: Applicative f =>
    --     (a -> f b) -> Maybe a -> f (Maybe b)
    traverse _ Nothing  = pure Nothing
    traverse g (Just x) = pure Just <*> g x
```

List traversable:

```
instance Traversable [] where
   -- traverse :: Applicative f => (a -> f b) -> [a] -> f [b]
   traverse _ []     = pure []
   traverse g (x:xs) = pure (:) <*> g x <*> traverse g xs
```