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 *n*th 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 *n*th 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 = ...
```