Appendix A

Standard prelude

In this appendix we present some of the most commonly used definitions from the standard prelude. For clarity, a number of the definitions have been simplified or modified from those given in the Haskell Report (25).

A.1 Classes

Equality types:

```
class Eq a where
  (==), (≠)        :: a → a → Bool

  x ≠ y            =  ¬ (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 (Eq a, Show a) ⇒ 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
```

Monadic types:

```
class Monad m where
  return                 :: a → m a
  (⩥)                    :: m a → (a → m b) → m b
```

A.2 Logical values

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:

```
¬                        :: Bool → Bool
¬ False                  =  True
¬ True                   =  False
```

Guard that always succeeds:

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

A.3 Characters and strings

Type declarations:

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

type String = [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
```

A.4 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)
```

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                       =  ¬ ∘ even
```

Exponentiation:

```
(↑)                       :: (Num a, Integral b) ⇒ a → b → a
_ ↑ 0                     =  1
x ↑ (n + 1)               =  x ∗ (x ↑ n)
```

A.5 | Tuples

Type declarations:

$$
\begin{array}{lcl}
\textbf{data}\ () & = & \cdots \\
 & & \textbf{deriving}\ (Eq, Ord, Show, Read) \\
\textbf{data}\ (a, b) & = & \cdots \\
 & & \textbf{deriving}\ (Eq, Ord, Show, Read) \\
\textbf{data}\ (a, b, c) & = & \cdots \\
 & & \textbf{deriving}\ (Eq, Ord, Show, Read) \\
\vdots & &
\end{array}
$$

Select the first component of a pair:

$$
\begin{array}{lcl}
fst & :: & (a, b) \to a \\
fst\ (x, \_) & = & x
\end{array}
$$

Select the second component of a pair:

$$
\begin{array}{lcl}
snd & :: & (a, b) \to b \\
snd\ (\_, y) & = & y
\end{array}
$$

A.6 | Maybe

Type declaration:

$$
\begin{array}{lcl}
\textbf{data}\ Maybe\ a & = & Nothing \mid Just\ a \\
 & & \textbf{deriving}\ (Eq, Ord, Show, Read)
\end{array}
$$

A.7 | Lists

Type declaration:

$$
\begin{array}{lcl}
\textbf{data}\ [a] & = & [\,] \mid a : [a] \\
 & & \textbf{deriving}\ (Eq, Ord, Show, Read)
\end{array}
$$

Decide if a list is empty:

$$
\begin{array}{lcl}
null & :: & [a] \to Bool \\
null\ [\,] & = & True \\
null\ (\_ : \_) & = & False
\end{array}
$$

Decide if a value is an element of a list:

$$
\begin{array}{lcl}
elem & :: & Eq\ a \Rightarrow a \to [a] \to Bool \\
elem\ x\ xs & = & any\ (== x)\ xs
\end{array}
$$

Decide if all logical values in a list are $True$:

$$
\begin{array}{lcl}
and & :: & [Bool] \to Bool
\end{array}
$$

```
and                 =  foldr (∧) True
```

Decide if any logical value in a list is *False*:

```
or                  :: [Bool] → Bool
or                  =  foldr (∨) False
```

Decide if all elements of a list satisfy a predicate:

```
all                 :: (a → Bool) → [a] → Bool
all p               =  and ∘ map p
```

Decide if any element of a list satisfies a predicate:

```
any                 :: (a → Bool) → [a] → Bool
any p               =  or ∘ map p
```

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 + 1) =  xs !! n
```

Select the first n elements of a list:

```
take                :: Int → [a] → [a]
take 0 _            =  []
take (n + 1) []     =  []
take (n + 1) (x : xs) =  x : take n 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 (n + 1) []        =  []
drop (n + 1) (_ : xs)  =  drop n 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)
```

Split a list using a predicate:

```
span                   ::  (a → Bool) → [a] → ([a], [a])
span p xs              =  (takeWhile p xs, dropWhile p xs)
```

Process a list using an operator that associates to the right:

```
foldr                  ::  (a → b → b) → b → [a] → b
foldr _ v []           =  v
foldr f v (x : xs)     =  f x (foldr f v xs)
```

Process a non-empty list using an operator that associates to the right:

```
foldr1                 ::  (a → a → a) → [a] → a
foldr1 _ [x]           =  x
foldr1 f (x : xs)      =  f x (foldr1 f xs)
```

Process a list using an operator that associates to the left:

```
foldl                  ::  (a → b → a) → a → [b] → a
foldl _ v []           =  v
foldl f v (x : xs)     =  foldl f (f v x) xs
```

Process a non-empty list using an operator that associates to the left:

```
foldl1                 ::  (a → a → a) → [a] → a
foldl1 f (x : xs)      =  foldl f x xs
```

Produce an infinite list of identical elements:

$$\begin{array}{lcl} repeat & :: & a \to [\,a\,] \\ repeat\ x & = & xs \ \textbf{where}\ xs = x : xs \end{array}$$

Produce a list with n identical elements:

$$\begin{array}{lcl} replicate & :: & Int \to a \to [\,a\,] \\ replicate\ n & = & take\ n \circ repeat \end{array}$$

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

$$\begin{array}{lcl} iterate & :: & (a \to a) \to a \to [\,a\,] \\ iterate\ f\ x & = & x : iterate\ f\ (f\ x) \end{array}$$

Produce a list of pairs from a pair of lists:

$$\begin{array}{lcl} zip & :: & [\,a\,] \to [\,b\,] \to [(a, b)] \\ zip\ [\,]\ \_ & = & [\,] \\ zip\ \_\ [\,] & = & [\,] \\ zip\ (x : xs)\ (y : ys) & = & (x, y) : zip\ xs\ ys \end{array}$$

Calculate the length of a list:

$$\begin{array}{lcl} length & :: & [\,a\,] \to Int \\ length & = & foldl\ (\lambda n\ \_ \to n + 1)\ 0 \end{array}$$

Calculate the sum of a list of numbers:

$$\begin{array}{lcl} sum & :: & Num\ a \Rightarrow [\,a\,] \to a \\ sum & = & foldl\ (+)\ 0 \end{array}$$

Calculate the product of a list of numbers:

$$\begin{array}{lcl} product & :: & Num\ a \Rightarrow [\,a\,] \to a \\ product & = & foldl\ (*)\ 1 \end{array}$$

Calculate the minimum of a non-empty list:

$$\begin{array}{lcl} minimum & :: & Ord\ a \Rightarrow [\,a\,] \to a \\ minimum & = & foldl1\ min \end{array}$$

Calculate the maximum of a non-empty list:

$$\begin{array}{lcl} maximum & :: & Ord\ a \Rightarrow [\,a\,] \to a \\ maximum & = & foldl1\ max \end{array}$$

Append two lists:

$$\begin{array}{lcl} (+\!\!+) & :: & [\,a\,] \to [\,a\,] \to [\,a\,] \\ [\,] +\!\!+ ys & = & ys \\ (x : xs) +\!\!+ ys & = & x : (xs +\!\!+ ys) \end{array}$$

Concatenate a list of lists:

$$\begin{array}{lcl} concat & :: & [[\,a\,]] \to [\,a\,] \\ concat & = & foldr\ (+\!\!+)\ [\,] \end{array}$$

Reverse a list:

$$\begin{array}{lcl} reverse & :: & [a] \to [a] \\ reverse & = & foldl\ (\lambda xs\ x \to x : xs)\ [\,] \end{array}$$

Apply a function to all elements of a list:

$$\begin{array}{lcl} map & :: & (a \to b) \to [a] \to [b] \\ map\ f\ xs & = & [f\ x \mid x \leftarrow xs] \end{array}$$

A.8 Functions

Type declaration:

$$\begin{array}{lcl} \textbf{data}\ a \to b & = & \cdots \end{array}$$

Identity function:

$$\begin{array}{lcl} id & :: & a \to a \\ id & = & \lambda x \to x \end{array}$$

Function composition:

$$\begin{array}{lcl} (\circ) & :: & (b \to c) \to (a \to b) \to (a \to c) \\ f \circ g & = & \lambda x \to f\ (g\ x) \end{array}$$

Constant functions:

$$\begin{array}{lcl} const & :: & a \to (b \to a) \\ const\ x & = & \lambda\_ \to x \end{array}$$

Strict application:

$$\begin{array}{lcl} (\$!) & :: & (a \to b) \to a \to b \\ f\ \$!\ x & = & \cdots \end{array}$$

Convert a function on pairs to a curried function:

$$\begin{array}{lcl} curry & :: & ((a, b) \to c) \to (a \to b \to c) \\ curry\ f & = & \lambda x\ y \to f\ (x, y) \end{array}$$

Convert a curried function to a function on pairs:

$$\begin{array}{lcl} uncurry & :: & (a \to b \to c) \to ((a, b) \to c) \\ uncurry\ f & = & \lambda(x, y) \to f\ x\ y \end{array}$$

A.9 Input/output

Type declaration:

$$\begin{array}{lcl} \textbf{data}\ IO\ a & = & \cdots \end{array}$$

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 = ...
```