# 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:

$$
\begin{array}{l}
\textbf{class}\ Eq\ a\ \textbf{where} \\
\quad (==), (\neq) \quad :: \quad a \to a \to Bool \\
\quad x \neq y \quad = \quad \neg\ (x == y)
\end{array}
$$

Ordered types:

$$
\begin{array}{l}
\textbf{class}\ Eq\ a \Rightarrow Ord\ a\ \textbf{where} \\
\quad (<), (\leq), (>), (\geq) \quad :: \quad a \to a \to Bool \\
\quad min, max \quad :: \quad a \to a \to a \\
\quad min\ x\ y \mid x \leq y \quad = \quad x \\
\qquad\qquad \mid otherwise \quad = \quad y \\
\quad max\ x\ y \mid x \leq y \quad = \quad y \\
\qquad\qquad \mid otherwise \quad = \quad x
\end{array}
$$

Showable types:

$$
\begin{array}{l}
\textbf{class}\ Show\ a\ \textbf{where} \\
\quad show \quad :: \quad a \to String
\end{array}
$$

Readable types:

$$
\begin{array}{l}
\textbf{class}\ Read\ a\ \textbf{where} \\
\quad read \quad :: \quad String \to a
\end{array}
$$

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:

$$\begin{array}{lcl} \textbf{data}\ \mathit{Char} & = & \cdots \\ & & \quad \textbf{deriving}\ (\mathit{Eq}, \mathit{Ord}, \mathit{Show}, \mathit{Read}) \end{array}$$

$$\textbf{type}\ \mathit{String} = [\mathit{Char}]$$

Decide if a character is a lower-case letter:

$$\begin{array}{lcl} \mathit{isLower} & :: & \mathit{Char} \to \mathit{Bool} \\ \mathit{isLower}\ c & = & c \geq \texttt{'a'} \wedge c \leq \texttt{'z'} \end{array}$$

Decide if a character is an upper-case letter:

$$\begin{array}{lcl} \mathit{isUpper} & :: & \mathit{Char} \to \mathit{Bool} \\ \mathit{isUpper}\ c & = & c \geq \texttt{'A'} \wedge c \leq \texttt{'Z'} \end{array}$$

Decide if a character is alphabetic:

$$\begin{array}{lcl} \mathit{isAlpha} & :: & \mathit{Char} \to \mathit{Bool} \\ \mathit{isAlpha}\ c & = & \mathit{isLower}\ c \vee \mathit{isUpper}\ c \end{array}$$

Decide if a character is a digit:

$$\begin{array}{lcl} \mathit{isDigit} & :: & \mathit{Char} \to \mathit{Bool} \\ \mathit{isDigit}\ c & = & c \geq \texttt{'0'} \wedge c \leq \texttt{'9'} \end{array}$$

Decide if a character is alpha-numeric:

$$\begin{array}{lcl} \mathit{isAlphaNum} & :: & \mathit{Char} \to \mathit{Bool} \\ \mathit{isAlphaNum}\ c & = & \mathit{isAlpha}\ c \vee \mathit{isDigit}\ c \end{array}$$

Decide if a character is spacing:

$$\begin{array}{lcl} \mathit{isSpace} & :: & \mathit{Char} \to \mathit{Bool} \\ \mathit{isSpace}\ c & = & \mathit{elem}\ c\ \texttt{" \t\n"} \end{array}$$

Convert a character to a Unicode number:

$$\begin{array}{lcl} \mathit{ord} & :: & \mathit{Char} \to \mathit{Int} \\ \mathit{ord}\ c & = & \cdots \end{array}$$

Convert a Unicode number to a character:

$$\begin{array}{lcl} \mathit{chr} & :: & \mathit{Int} \to \mathit{Char} \\ \mathit{chr}\ n & = & \cdots \end{array}$$

Convert a digit to an integer:

$$\begin{array}{lcl} \mathit{digitToInt} & :: & \mathit{Char} \to \mathit{Int} \\ \mathit{digitToInt}\ c \mid \mathit{isDigit}\ c & = & \mathit{ord}\ c - \mathit{ord}\ \texttt{'0'} \end{array}$$

Convert an integer to a digit:

$$
\begin{array}{lcl}
intToDigit & :: & Int \to Char \\
intToDigit\ n & & \\
\quad \mid n \geq 0 \wedge n \leq 9 & = & chr\ (ord\ \texttt{'0'} + n)
\end{array}
$$

Convert a letter to lower-case:

$$
\begin{array}{lcl}
toLower & :: & Char \to Char \\
toLower\ c \mid isUpper\ c & = & chr\ (ord\ c - ord\ \texttt{'A'} + ord\ \texttt{'a'}) \\
\quad \mid otherwise & = & c
\end{array}
$$

Convert a letter to upper-case:

$$
\begin{array}{lcl}
toUpper & :: & Char \to Char \\
toUpper\ c \mid isLower\ c & = & chr\ (ord\ c - ord\ \texttt{'a'} + ord\ \texttt{'A'}) \\
\quad \mid otherwise & = & c
\end{array}
$$

## A.4 Numbers

Type declarations:

$$
\begin{array}{lcl}
\textbf{data}\ Int & = & \cdots \\
 & & \textbf{deriving}\ (Eq, Ord, Show, Read, \\
 & & \qquad Num, Integral) \\
\textbf{data}\ Integer & = & \cdots \\
 & & \textbf{deriving}\ (Eq, Ord, Show, Read, \\
 & & \qquad Num, Integral) \\
\textbf{data}\ Float & = & \cdots \\
 & & \textbf{deriving}\ (Eq, Ord, Show, Read, \\
 & & \qquad Num, Fractional)
\end{array}
$$

Decide if an integer is even:

$$
\begin{array}{lcl}
even & :: & Integral\ a \Rightarrow a \to Bool \\
even\ n & = & n\ `mod`\ 2 == 0
\end{array}
$$

Decide if an integer is odd:

$$
\begin{array}{lcl}
odd & :: & Integral\ a \Rightarrow a \to Bool \\
odd & = & \neg \circ even
\end{array}
$$

Exponentiation:

$$
\begin{array}{lcl}
(\uparrow) & :: & (Num\ a, Integral\ b) \Rightarrow a \to b \to a \\
\_ \uparrow 0 & = & 1 \\
x \uparrow (n+1) & = & x * (x \uparrow n)
\end{array}
$$

## A.5 | 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
```

## A.6 | Maybe

Type declaration:

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

## A.7 | Lists

Type declaration:

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

Decide if a list is empty:

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

Decide if a value is an element of a list:

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

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

```
and                     :: [Bool] -> Bool
```

```
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 $n$th 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 $n$th 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:

```
reverse          :: [a] → [a]
reverse          =  foldl (λxs x → x : xs) [ ]
```

Apply a function to all elements of a list:

```
map              :: (a → b) → [a] → [b]
map f xs         =  [f x | x ← xs]
```

## A.8 Functions

Type declaration:

```
data a → b       =  ···
```

Identity function:

```
id               :: a → a
id               =  λx → x
```

Function composition:

```
(∘)              :: (b → c) → (a → b) → (a → c)
f ∘ g            =  λx → f (g x)
```

Constant functions:

```
const            :: a → (b → a)
const x          =  λ_ → x
```

Strict application:

```
($!)             :: (a → b) → a → b
f $! x           =  ···
```

Convert a function on pairs to a curried function:

```
curry            :: ((a, b) → c) → (a → b → c)
curry f          =  λx y → f (x, y)
```

Convert a curried function to a function on pairs:

```
uncurry          :: (a → b → c) → ((a, b) → c)
uncurry f        =  λ(x, y) → f x y
```

## A.9 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 = ...
```