

---

# 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:

$$\mathbf{data} \textit{Char} \quad = \dots$$

$$\mathbf{deriving} (Eq, Ord, Show, Read)$$

$$\mathbf{type} \textit{String} \quad = [\textit{Char}]$$

Decide if a character is a lower-case letter:

$$\textit{isLower} \quad :: \textit{Char} \rightarrow \textit{Bool}$$

$$\textit{isLower} \ c \quad = c \geq 'a' \wedge c \leq 'z'$$

Decide if a character is an upper-case letter:

$$\textit{isUpper} \quad :: \textit{Char} \rightarrow \textit{Bool}$$

$$\textit{isUpper} \ c \quad = c \geq 'A' \wedge c \leq 'Z'$$

Decide if a character is alphabetic:

$$\textit{isAlpha} \quad :: \textit{Char} \rightarrow \textit{Bool}$$

$$\textit{isAlpha} \ c \quad = \textit{isLower} \ c \vee \textit{isUpper} \ c$$

Decide if a character is a digit:

$$\textit{isDigit} \quad :: \textit{Char} \rightarrow \textit{Bool}$$

$$\textit{isDigit} \ c \quad = c \geq '0' \wedge c \leq '9'$$

Decide if a character is alpha-numeric:

$$\textit{isAlphaNum} \quad :: \textit{Char} \rightarrow \textit{Bool}$$

$$\textit{isAlphaNum} \ c \quad = \textit{isAlpha} \ c \vee \textit{isDigit} \ c$$

Decide if a character is spacing:

$$\textit{isSpace} \quad :: \textit{Char} \rightarrow \textit{Bool}$$

$$\textit{isSpace} \ c \quad = \textit{elem} \ c \ " \ \backslash \textit{t} \backslash \textit{n} \textit{"}$$

Convert a character to a Unicode number:

$$\textit{ord} \quad :: \textit{Char} \rightarrow \textit{Int}$$

$$\textit{ord} \ c \quad = \dots$$

Convert a Unicode number to a character:

$$\textit{chr} \quad :: \textit{Int} \rightarrow \textit{Char}$$

$$\textit{chr} \ n \quad = \dots$$

Convert a digit to an integer:

$$\textit{digitToInt} \quad :: \textit{Char} \rightarrow \textit{Int}$$

$$\textit{digitToInt} \ c \mid \textit{isDigit} \ c \quad = \textit{ord} \ c - \textit{ord} \ '0'$$

Convert an integer to a digit:

$$\begin{aligned} \text{intToDigit} &:: \text{Int} \rightarrow \text{Char} \\ \text{intToDigit } n &= \text{chr } (\text{ord } '0' + n) \\ &| n \geq 0 \wedge n \leq 9 \end{aligned}$$

Convert a letter to lower-case:

$$\begin{aligned} \text{toLower} &:: \text{Char} \rightarrow \text{Char} \\ \text{toLower } c \mid \text{isUpper } c &= \text{chr } (\text{ord } c - \text{ord } 'A' + \text{ord } 'a') \\ &| \text{otherwise} &= c \end{aligned}$$

Convert a letter to upper-case:

$$\begin{aligned} \text{toUpper} &:: \text{Char} \rightarrow \text{Char} \\ \text{toUpper } c \mid \text{isLower } c &= \text{chr } (\text{ord } c - \text{ord } 'a' + \text{ord } 'A') \\ &| \text{otherwise} &= c \end{aligned}$$

## A.4 Numbers

Type declarations:

$$\begin{aligned} \mathbf{data} \text{ Int} &= \dots \\ &\quad \mathbf{deriving} (\text{Eq}, \text{Ord}, \text{Show}, \text{Read}, \\ &\quad \quad \text{Num}, \text{Integral}) \\ \mathbf{data} \text{ Integer} &= \dots \\ &\quad \mathbf{deriving} (\text{Eq}, \text{Ord}, \text{Show}, \text{Read}, \\ &\quad \quad \text{Num}, \text{Integral}) \\ \mathbf{data} \text{ Float} &= \dots \\ &\quad \mathbf{deriving} (\text{Eq}, \text{Ord}, \text{Show}, \text{Read}, \\ &\quad \quad \text{Num}, \text{Fractional}) \end{aligned}$$

Decide if an integer is even:

$$\begin{aligned} \text{even} &:: \text{Integral } a \Rightarrow a \rightarrow \text{Bool} \\ \text{even } n &= n \text{ 'mod' } 2 == 0 \end{aligned}$$

Decide if an integer is odd:

$$\begin{aligned} \text{odd} &:: \text{Integral } a \Rightarrow a \rightarrow \text{Bool} \\ \text{odd} &= \neg \circ \text{even} \end{aligned}$$

Exponentiation:

$$\begin{aligned} (\uparrow) &:: (\text{Num } a, \text{Integral } b) \Rightarrow a \rightarrow b \rightarrow a \\ - \uparrow 0 &= 1 \\ x \uparrow (n + 1) &= x * (x \uparrow n) \end{aligned}$$

---

## 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

```

$$\text{and} \quad = \text{foldr } (\wedge) \text{ True}$$

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

$$\begin{aligned} \text{or} & \quad :: [\text{Bool}] \rightarrow \text{Bool} \\ \text{or} & \quad = \text{foldr } (\vee) \text{ False} \end{aligned}$$

Decide if all elements of a list satisfy a predicate:

$$\begin{aligned} \text{all} & \quad :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool} \\ \text{all } p & \quad = \text{and} \circ \text{map } p \end{aligned}$$

Decide if any element of a list satisfies a predicate:

$$\begin{aligned} \text{any} & \quad :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool} \\ \text{any } p & \quad = \text{or} \circ \text{map } p \end{aligned}$$

Select the first element of a non-empty list:

$$\begin{aligned} \text{head} & \quad :: [a] \rightarrow a \\ \text{head } (x : \_) & \quad = x \end{aligned}$$

Select the last element of a non-empty list:

$$\begin{aligned} \text{last} & \quad :: [a] \rightarrow a \\ \text{last } [x] & \quad = x \\ \text{last } (\_ : xs) & \quad = \text{last } xs \end{aligned}$$

Select the *n*th element of a non-empty list:

$$\begin{aligned} (!!) & \quad :: [a] \rightarrow \text{Int} \rightarrow a \\ (x : \_) !! 0 & \quad = x \\ (\_ : xs) !! (n + 1) & \quad = xs !! n \end{aligned}$$

Select the first *n* elements of a list:

$$\begin{aligned} \text{take} & \quad :: \text{Int} \rightarrow [a] \rightarrow [a] \\ \text{take } 0 \_ & \quad = [] \\ \text{take } (n + 1) [] & \quad = [] \\ \text{take } (n + 1) (x : xs) & \quad = x : \text{take } n \ xs \end{aligned}$$

Select all elements of a list that satisfy a predicate:

$$\begin{aligned} \text{filter} & \quad :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\ \text{filter } p \ xs & \quad = [x \mid x \leftarrow xs, p \ x] \end{aligned}$$

Select elements of a list while they satisfy a predicate:

$$\begin{aligned} \text{takeWhile} & \quad :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\ \text{takeWhile } \_ [] & \quad = [] \\ \text{takeWhile } p \ (x : xs) & \quad = \begin{cases} p \ x & = x : \text{takeWhile } p \ xs \\ \text{otherwise} & = [] \end{cases} \end{aligned}$$

Remove the first element from a non-empty list:

$$\text{tail} \quad :: [a] \rightarrow [a]$$





Reverse a list:

$$\begin{aligned} \textit{reverse} &:: [a] \rightarrow [a] \\ \textit{reverse} &= \textit{foldl} (\lambda xs\ x \rightarrow x : xs) [] \end{aligned}$$

Apply a function to all elements of a list:

$$\begin{aligned} \textit{map} &:: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\ \textit{map}\ f\ xs &= [f\ x \mid x \leftarrow xs] \end{aligned}$$

---

## A.8 | Functions

Type declaration:

$$\mathbf{data}\ a \rightarrow b \quad = \dots$$

Identity function:

$$\begin{aligned} \textit{id} &:: a \rightarrow a \\ \textit{id} &= \lambda x \rightarrow x \end{aligned}$$

Function composition:

$$\begin{aligned} (\circ) &:: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \\ f \circ g &= \lambda x \rightarrow f\ (g\ x) \end{aligned}$$

Constant functions:

$$\begin{aligned} \textit{const} &:: a \rightarrow (b \rightarrow a) \\ \textit{const}\ x &= \lambda _ \rightarrow x \end{aligned}$$

Strict application:

$$\begin{aligned} (\$!) &:: (a \rightarrow b) \rightarrow a \rightarrow b \\ f\ \$!\ x &= \dots \end{aligned}$$

Convert a function on pairs to a curried function:

$$\begin{aligned} \textit{curry} &:: ((a, b) \rightarrow c) \rightarrow (a \rightarrow b \rightarrow c) \\ \textit{curry}\ f &= \lambda x\ y \rightarrow f\ (x, y) \end{aligned}$$

Convert a curried function to a function on pairs:

$$\begin{aligned} \textit{uncurry} &:: (a \rightarrow b \rightarrow c) \rightarrow ((a, b) \rightarrow c) \\ \textit{uncurry}\ f &= \lambda (x, y) \rightarrow f\ x\ y \end{aligned}$$

---

## A.9 | Input/output

Type declaration:

$$\mathbf{data}\ IO\ a \quad = \dots$$

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     = ...

```