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:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class } Eq\ a \textbf{ where}} \\
\quad (==), (\neq) & :: & a \to a \to Bool \\
\quad \overset{/=}{x \neq y} & = & \overset{\text{not}}{\neg}\,(x == y)
\end{array}
$$

Ordered types:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class } Eq\ a \Rightarrow Ord\ a \textbf{ where}} \\
\quad (<), (\leq), (>), (\geq) & :: & a \to a \to Bool \\
\quad min, max & :: & a \to a \to a \\
\\
\quad min\ x\ y \mid x \leq y & = & x \\
\quad \qquad\quad\ \mid otherwise & = & y \\
\quad max\ x\ y \mid x \leq y & = & y \\
\quad \qquad\quad\ \mid otherwise & = & x
\end{array}
$$

Showable types:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class } Show\ a \textbf{ where}} \\
\quad show & :: & a \to String
\end{array}
$$

Readable types:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class } Read\ a \textbf{ where}} \\
\quad read & :: & String \to a
\end{array}
$$

Numeric types:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class } (Eq\ a, Show\ a) \Rightarrow Num\ a \textbf{ where}} \\
\quad (+), (-), (*) & :: & a \to a \to a \\
\quad negate, abs, signum & :: & a \to a
\end{array}
$$

Integral types:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class } Num\ a \Rightarrow Integral\ a \textbf{ where}} \\
\quad div, mod & :: & a \to a \to a
\end{array}
$$

Fractional types:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class } Num\ a \Rightarrow Fractional\ a \textbf{ where}} \\
\quad (/) & :: & a \to a \to a \\
\quad recip & :: & a \to a \\
\\
\quad recip\ n & = & 1\ /\ n
\end{array}
$$

Monadic types:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{class } Monad\ m \textbf{ where}} \\
\quad return & :: & a \to m\ a \\
\quad (\gg=) & :: & m\ a \to (a \to m\ b) \to m\ b
\end{array}
$$

A.2 | Logical values

Type declaration:

$$
\begin{array}{lll}
\textbf{data } Bool & = & False \mid True \\
 & & \textbf{deriving } (Eq, Ord, Show, Read)
\end{array}
$$

Logical conjunction:

$$
\begin{array}{lll}
(\wedge) & :: & Bool \to Bool \to Bool \\
False \wedge \_ & = & False \\
True \wedge b & = & b
\end{array}
$$

Logical disjunction:

$$
\begin{array}{lll}
(\vee) & :: & Bool \to Bool \to Bool \\
False \vee b & = & b \\
True \vee \_ & = & True
\end{array}
$$

Logical negation:

$$
\begin{array}{lll}
\neg & :: & Bool \to Bool \\
\neg\, False & = & True \\
\neg\, True & = & False
\end{array}
$$

Guard that always succeeds:

$$
\begin{array}{lll}
otherwise & :: & Bool \\
otherwise & = & True
\end{array}
$$

A.3 Characters and strings

Type declarations:

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

Decide if a character is a lower-case letter:

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

Decide if a character is an upper-case letter:

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

Decide if a character is alphabetic:

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

Decide if a character is a digit:

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

Decide if a character is alpha-numeric:

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

Decide if a character is spacing:

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

Convert a character to a Unicode number:

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

Convert a Unicode number to a character:

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

Convert a digit to an integer:

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

Convert an integer to a digit:

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

Convert a letter to lower-case:

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

Convert a letter to upper-case:

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

A.4 Numbers

Type declarations:

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

Decide if an integer is even:

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

Decide if an integer is odd:

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

Exponentiation:

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

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:

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

Select the second component of a pair:

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

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:

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

Decide if a value is an element of a list:

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

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

$$\begin{array}{lcl}
and & :: & [Bool] \rightarrow Bool \\
and & = & foldr\ (\wedge)\ True
\end{array}$$

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

$$\begin{array}{lcl}
or & :: & [Bool] \rightarrow Bool \\
or & = & foldr\ (\vee)\ False
\end{array}$$

Decide if all elements of a list satisfy a predicate:

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

Decide if any element of a list satisfies a predicate:

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

Select the first element of a non-empty list:

$$\begin{array}{lcl}
head & :: & [a] \rightarrow a \\
head\ (x : _) & = & x
\end{array}$$

Select the last element of a non-empty list:

$$\begin{array}{lcl}
last & :: & [a] \rightarrow a \\
last\ [x] & = & x \\
last\ (_ : xs) & = & last\ xs
\end{array}$$

Select the nth element of a non-empty list:

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

Select the first n elements of a list:

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

Select all elements of a list that satisfy a predicate:

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

Select elements of a list while they satisfy a predicate:

$$\begin{array}{lcl}
takeWhile & :: & (a \rightarrow Bool) \rightarrow [a] \rightarrow [a] \\
takeWhile\ _\ [\,] & = & [\,] \\
takeWhile\ p\ (x : xs) & & \\
\quad \mid p\ x & = & x : takeWhile\ p\ xs \\
\quad \mid otherwise & = & [\,]
\end{array}$$

Remove the first element from a non-empty list:

$$\begin{array}{lcl}
tail & :: & [a] \rightarrow [a]
\end{array}$$

$$\begin{array}{lcl} \mathit{tail}\ (\_ : \mathit{xs}) & = & \mathit{xs} \end{array}$$

Remove the last element from a non-empty list:

$$\begin{array}{lcl} \mathit{init} & :: & [a] \to [a] \\ \mathit{init}\ [\_] & = & [\,] \\ \mathit{init}\ (x : \mathit{xs}) & = & x : \mathit{init}\ \mathit{xs} \end{array}$$

Remove the first n elements from a list:

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

Remove elements from a list while they satisfy a predicate:

$$\begin{array}{lcl} \mathit{dropWhile} & :: & (a \to \mathit{Bool}) \to [a] \to [a] \\ \mathit{dropWhile}\ \_\ [\,] & = & [\,] \\ \mathit{dropWhile}\ p\ (x : \mathit{xs}) & & \\ \quad \mid p\ x & = & \mathit{dropWhile}\ p\ \mathit{xs} \\ \quad \mid \mathit{otherwise} & = & x : \mathit{xs} \end{array}$$

Split a list at the nth element:

$$\begin{array}{lcl} \mathit{splitAt} & :: & \mathit{Int} \to [a] \to ([a],[a]) \\ \mathit{splitAt}\ n\ \mathit{xs} & = & (\mathit{take}\ n\ \mathit{xs}, \mathit{drop}\ n\ \mathit{xs}) \end{array}$$

Split a list using a predicate:

$$\begin{array}{lcl} \mathit{span} & :: & (a \to \mathit{Bool}) \to [a] \to ([a],[a]) \\ \mathit{span}\ p\ \mathit{xs} & = & (\mathit{takeWhile}\ p\ \mathit{xs}, \mathit{dropWhile}\ p\ \mathit{xs}) \end{array}$$

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

$$\begin{array}{lcl} \mathit{foldr} & :: & (a \to b \to b) \to b \to [a] \to b \\ \mathit{foldr}\ \_\ v\ [\,] & = & v \\ \mathit{foldr}\ f\ v\ (x : \mathit{xs}) & = & f\ x\ (\mathit{foldr}\ f\ v\ \mathit{xs}) \end{array}$$

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

$$\begin{array}{lcl} \mathit{foldr1} & :: & (a \to a \to a) \to [a] \to a \\ \mathit{foldr1}\ \_\ [x] & = & x \\ \mathit{foldr1}\ f\ (x : \mathit{xs}) & = & f\ x\ (\mathit{foldr1}\ f\ \mathit{xs}) \end{array}$$

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

$$\begin{array}{lcl} \mathit{foldl} & :: & (a \to b \to a) \to a \to [b] \to a \\ \mathit{foldl}\ \_\ v\ [\,] & = & v \\ \mathit{foldl}\ f\ v\ (x : \mathit{xs}) & = & \mathit{foldl}\ f\ (f\ v\ x)\ \mathit{xs} \end{array}$$

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

$$\begin{array}{lcl} \mathit{foldl1} & :: & (a \to a \to a) \to [a] \to a \\ \mathit{foldl1}\ f\ (x : \mathit{xs}) & = & \mathit{foldl}\ f\ x\ \mathit{xs} \end{array}$$

Produce an infinite list of identical elements:

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

Produce a list with n identical elements:

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

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

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

Produce a list of pairs from a pair of lists:

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

Calculate the length of a list:

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

Calculate the sum of a list of numbers:

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

Calculate the product of a list of numbers:

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

Calculate the minimum of a non-empty list:

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

Calculate the maximum of a non-empty list:

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

Append two lists:

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

Concatenate a list of lists:

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

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
```

right-associative

Application

($) :: (a→b) → a → b

f $ x = f 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       =  · · ·
```