Exercises from Chapter 1

Exercise 1.4

The replacement of the condition $k \uparrow 2 > n$ by $k \uparrow 2 \geqslant n$ in the definition of *ldf* makes no difference. This is because, in the case where k is such that $k \uparrow 2 = n$, the condition *divides* k n is true. Thus the case is handled by the first guarded equation.

Exercise 1.6

Based only on *divides*, I would guess:

$$\mathit{rem} :: \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer}$$

Exercise 1.7

divides 5 is a procedure that takes an integer as a parameter and returns a boolean value. Thus it is an expression of type $\mathit{Integer} \to \mathit{Bool}$.

divides 5 7 evaluates to a boolean value, thus it is an expression of type *Bool*.

Exercise 1.9

A function that gives the maximum of a list of integers, using the predefined function *max*.

$$\begin{array}{ll}
\mathit{maxInt} & :: [\mathit{Int}] \to \mathit{Int} \\
\mathit{maxInt}\ [\,] & = \mathit{error}\ \texttt{"empty list"} \\
\mathit{maxInt}\ [x] & = x \\
\mathit{maxInt}\ (x : \mathit{xs}) & = \mathit{max}\ x\ (\mathit{maxInt}\ \mathit{xs})
\end{array}$$

Exercise 1.10

A function *removeFst* that removes the first occurrence of an integer m from a list of integers. If m does not occur in the list, the list remains unchanged.

$$\begin{array}{llll}
\mathit{removeFst} & :: \mathit{Int} \to [\mathit{Int}] \to [\mathit{Int}] & & \\
\mathit{removeFst}\ _\ [\,] & = [\,] & & \\
\mathit{removeFst}\ m\ (x : \mathit{xs}) & \mid\ m \equiv x & = \mathit{xs} & \\
 & \mid\ \mathit{otherwise} & = x : \mathit{removeFst}\ m\ \mathit{xs} &
\end{array}$$

Exercise 1.13

A function *count* for counting the number of occurences of a character in a string.

$$
\begin{array}{lll}
\mathit{count} & :: \mathit{Char} \to \mathit{String} \to \mathit{Int} & \\
\mathit{count}\ \_\ [\,] & = 0 & \\
\mathit{count}\ c\ (x : \mathit{xs}) & \mid c \equiv x & = 1 + \mathit{count}\ c\ \mathit{xs} \\
 & \mid \mathit{otherwise} & = \mathit{count}\ c\ \mathit{xs}
\end{array}
$$

Exercise 1.14

A function *blowup* that converts a string $a_1, a_2, a_3, \ldots$ to $a_1, a_2, a_2, a_3, a_3, a_3 \ldots$

Using explicit recursion:

$$
\begin{array}{l}
\mathit{blowup} :: \mathit{String} \to \mathit{String} \\
\mathit{blowup} = \mathit{blowHelper}\ 1 \\
\quad \textbf{where} \\
\qquad \begin{array}{ll}
\mathit{blowHelper} & :: \mathit{Int} \to \mathit{String} \to \mathit{String} \\
\mathit{blowHelper}\ \_\ [\,] & = [\,] \\
\mathit{blowHelper}\ n\ (x : \mathit{xs}) & = \mathit{replicate}\ n\ x +\!\!+\ \mathit{blowHelper}\ (n + 1)\ \mathit{xs}
\end{array}
\end{array}
$$

More elegant solution:

$$
\begin{array}{l}
\mathit{blowup}' :: \mathit{String} \to \mathit{String} \\
\mathit{blowup}' = \mathit{concat} \circ (\mathit{zipWith}\ \mathit{replicate}\ [1\,..])
\end{array}
$$

Exercise 1.15

A function $\mathit{srtString} :: [\mathit{String}] \to [\mathit{String}]$ that sorts a list of strings in alphabetical order

$$
\begin{array}{llll}
\mathit{srtString} & & :: [\mathit{String}] \to [\mathit{String}] & \\
\mathit{srtString} & [\,] & = [\,] & \\
\mathit{srtString} & \mathit{xs} & = m : (\mathit{srtString}\ (\mathit{removeString}\ m\ \mathit{xs})) & \\
\multicolumn{4}{l}{\quad \textbf{where}\ m = \mathit{mnmString}\ \mathit{xs}} \\
\mathit{removeString} & & :: \mathit{String} \to [\mathit{String}] \to [\mathit{String}] & \\
\mathit{removeString}\ \_ & [\,] & = [\,] & \\
\mathit{removeString}\ m & (x : \mathit{xs}) & \mid m \equiv x & = \mathit{xs} \\
 & & \mid \mathit{otherwise} & = x : \mathit{removeString}\ m\ \mathit{xs} \\
\mathit{mnmString} & & :: [\mathit{String}] \to \mathit{String} & \\
\mathit{mnmString} & [\,] & = [\,] & \\
\mathit{mnmString} & [x] & = x & \\
\mathit{mnmString} & (x : \mathit{xs}) & = \mathit{min}\ x\ (\mathit{mnmString}\ \mathit{xs}) &
\end{array}
$$

Exercise 1.17

A function $substring :: String \to String \to Bool$ that checks whether *str1* is a substring of *str2*.

$$
\begin{array}{lll}
prefix & & :: String \to String \to Bool \\
prefix\ [\,] & ys & = True \\
prefix\ (x : xs)\ [\,] & & = False \\
prefix\ (x : xs)\ (y : ys) & & = (x \equiv y) \wedge prefix\ xs\ ys
\end{array}
$$

$$
\begin{array}{llll}
substring & & :: String \to String \to Bool & \\
substring\ [\,] & \_ & = True & \\
substring\ \_ & [\,] & = False & \\
substring\ xs & (y : ys) & \mid prefix\ xs\ (y : ys) & = True \\
 & & \mid substring\ xs\ ys & = True \\
 & & \mid otherwise & = False
\end{array}
$$

Exercise 1.18

Find expressions with the following types:

1. $[String]$
   answer: `["Text","More text"]`
2. $(Bool, String)$
   answer: ($True$, `"Indeed"`)
3. $[(Bool, String)]$
   answer: [($True$, `"Yes"`), ($False$, `"No"`)]
4. $([Bool], String)$
   answer: ([$True$, $False$], `"What?"`)
5. $Bool \to Bool$
   answer: $\neg$

Exercise 1.19

Find the types of the following predefined functions, supply them with arguments of the expected types, and try to guess what they do.

1. $head :: [a] \to a$
   returns the first element from a list
2. $last :: [a] \to a$
   returns the last element from a list

3. $init :: [a] \to [a]$
   returns a new list with the last element from the old list removed

4. $fst :: (a, b) \to a$
   returns the first element of a pair

5. $(+\!\!+) :: [a] \to [a] \to [a]$
   returns a list with the contents of the second list appended to the end of the first list

6. $flip :: (a \to b \to c) \to b \to a \to c$
   given a function of 2 arguments, returns a function with the order of the two arguments interchanged

7. $flip\ (+\!\!+) :: [a] \to [a] \to [a]$
   returns a list with the contents of the first list appended to the end of the second list

Exercise 1.20

Use *map* to write a function *lengths* that takes a list of lists and returns a list of the corresponding list lengths.

$$\begin{aligned} &lengths :: [[a]] \to [Int] \\ &lengths = map\ length \end{aligned}$$

Exercise 1.21

Use *map* to write a function *sumLengths* that takes a list of lists and returns the sum of their lenghts.

$$\begin{aligned} &sumLengths :: [[a]] \to Int \\ &sumLengths = sum \circ map\ length \end{aligned}$$

Exercise 1.22

We modify the defining equation of *ldp* as follows:

$$\begin{aligned} &ldp :: Integer \to Integer \\ &ldp = ldpf\ primes1 \end{aligned}$$

Now, *ldp* works exactly as though we had written $ldp\ n = ldpf\ primes1\ n$.

ldpf is a function of type $[Integer] \to Integer \to Integer$. What we did above was to supply it with only a first argument, causing it to return a function of type $Integer \to Integer$ – which is what we wanted for *ldp*.