Exercise 4.1
Assume the following Haskell definitions:

\[
\begin{align*}
\text{size} &= 10 \\
\text{square } n &= n \times n
\end{align*}
\]

Add a definition for \textit{cube} with the obvious meaning, and manually perform single-stepped expression evaluation for the expression “\textit{cube size - cube (size - 2)}”.

Solution Hints
\[
\text{cube } n = n \times \text{square } n
\]
Then:

\[
\begin{align*}
\text{cube size} - \text{cube (size - 2)} &= (\text{size} \times \text{square } \text{size}) - \text{cube (size - 2)} \quad \text{-- unfolding cube definition} \\
&= (10 \times \text{square } 10) - \text{cube (10 - 2)} \quad \text{-- unfolding size definition} \\
&= (10 \times (10 \times 10)) - \text{cube (10 - 2)} \quad \text{-- unfolding square definition} \\
&= (10 \times 100) - \text{cube (10 - 2)} \quad \text{-- multiplication} \\
&= 1000 - \text{cube (10 - 2)} \quad \text{-- multiplication} \\
&= 1000 - (10 - 2) \times \text{square (10 - 2)} \quad \text{-- unfolding cube definition} \\
&= 1000 - 8 \times \text{square } 8 \quad \text{-- subtraction} \\
&= 1000 - 8 \times (8 \times 8) \quad \text{-- unfolding square definition} \\
&= 1000 - 8 \times 64 \quad \text{-- multiplication} \\
&= 1000 - 512 \quad \text{-- multiplication} \\
&= 488 \quad \text{-- subtraction}
\end{align*}
\]

Exercise 4.2
Haskell has predefined types \textit{Float} for single-precision floating point numbers (which we ignore in the following) and \textit{Double} for double-precision floating point numbers.

Standard mathematical functions like

\[
\begin{align*}
\text{sqrt } , \text{sin } , \text{atan} :: \text{Double} \rightarrow \text{Double}
\end{align*}
\]

and \textit{pi} :: \text{Double} are also available; \textit{x^k} stands for \textit{x}^\textit{k} if \textit{k} is natural; \textit{x ** q} can be used for \textit{x}^\textit{q} where both \textit{x} and \textit{q} are of type \textit{Double}.

Define the following Haskell functions, with the meanings obvious from their names:

(a) \text{sphereVolume} :: \text{Double} \rightarrow \text{Double}

(b) \text{sphereSurface} :: \text{Double} \rightarrow \text{Double}

(c) \text{centuryToPicosecond} :: \text{Integer} \rightarrow \text{Integer}
Try the last one in C or Java, too; test both, and compare the results

**Solution Hints**
Introduce auxiliary constants or functions at least for (c)!

```haskell
sphereVolume :: Double -> Double
sphereVolume r = 4/3 * pi * r ^ 3

sphereSurface :: Double -> Double
sphereSurface r = 4 * pi * r^2

centuryToPicosecond :: Integer -> Integer
centuryToPicosecond c = c * daysPerCentury * 24 * 3600 * 10 ^ 12

daysPerCentury, daysPerYear, leapYearsPerCentury :: Integer
daysPerCentury = 100 * daysPerYear + leapYearsPerCentury
leapYearsPerCentury = 24

daysPerYear = 365

(This does not take leap-seconds into account.)

In C or Java, some extra effort would be required to make this work with some integral type, since:

```
Main> centuryToPicosecond 1
3155673600000000000000
Main> 2 ^ 64
18446744073709551616
```

---

**Exercise 4.3**
Define the following Haskell functions:

(a)  `stutter :: [a] -> [a]`
    duplicates each element of its argument lists, e.g.:  
    `stutter [1,2,3] = [1,1,2,2,3,3]`

**Solution Hints**
    ```haskell
    stutter :: [a] -> [a]
    stutter [] = []
    stutter (x:xs) = x : x : stutter xs
    ```

(b)  `splits :: [a] -> [( [a],[a])]`
    delivers for each argument list all possibilities to segment it into non-empty prefix and suffix, e.g.:  
    `splits [1,2,3] = [( [1],[2,3]), ([1,2],[3])]`
    (The order is irrelevant.)

**Solution Hints**
    ```haskell
    -- most “natural”:
    splits [] = []
    ```
splits \([x] = []\)
splits \((x:xs) = ([x],xs) : \text{map (pupd1 (x:)) (splits xs)}\)

\[= ([x],xs) : [ (x:pre, suff) \mid (pre,suff) \leftarrow \text{splits xs} ]\]

\[\text{pupd1 } f \ (x,y) = (f \ x, y)\]

\(--\ much\ less\ efficient:\)
splits' \([\ ] = \[]\)
splits' \((x : xs) = \text{spl} \ [x] \ xs\)

where
\[\text{spl} \ ys \ [\ ] = \[]\]
\[\text{spl} \ ys \ (xs@(x : xs')) = (ys, xs) : \text{spl} \ (ys ++ \ [x]) \ xs'\]

\(--\ roughly\ equally\ inefficient:\)
splits'' \(xs = \text{map (flip splitAt xs)} \ [1 \ldots \text{length } xs - 1]\)

(c) \(\text{rotations} :: [a] \rightarrow [[a]]\)
delivers for each argument list all different results of rotations, each result only once, e.g.:
\(\text{rotations} \ [1.2.3] = [[1.2.3], [3.1.2], [2.3.1]]\)
(The order is irrelevant.)

\[\text{Solution Hints}\]
\(\text{rotations} :: [a] \rightarrow [[a]]\)
\(\text{rotations} \ xs \ = \ xs : \text{map (uncurry (flip (++))) (splits xs)}\)

\[= xs : [ \text{suff ++ pre} \mid (pre, \text{suff}) \leftarrow \text{splits xs} ]\]

\(\text{rotations' } xs = \text{r} \ [\ ] \ xs\)

where
\[\text{r} \ ys \ [\ ] = [ys]\]
\[\text{r} \ ys \ xs@(x : xs') = (xs ++ ys) : \text{r} \ (ys ++ \ [x]) \ xs'\]

(d) \(\text{permutations} :: [a] \rightarrow [[a]]\)
delivers for each argument list all different results of permutations, each result only once, e.g.:
\(\text{permutations} \ [1.2.3] = [[1.2.3], [1.3.2], [2.1.3], [2.3.1], [3.1.2], [3.2.1]]\)
(The order is irrelevant.)

\[\text{Solution Hints}\]
\(\text{permutations} :: [a] \rightarrow [[a]]\)
\(\text{permutations} \ [\ ] = [[\ ]]\)
\(\text{permutations} \ xs =\)
\(\text{concat} \ [ \text{map (y:) (permutations } ys) \mid (y : ys) \leftarrow \text{rotations } xs ]\)

\(\text{permutations' } [\ ] = [[\ ]]\)
permutations' xs = concatMap permAux (rotations xs)
where
    permAux (y : ys) = map (y:) (permutations ys)

Exercise 4.4 — Defining Haskell Functions (40% of Midterm 1, 2003)
Define the following Haskell functions (the solutions are independent of each other):

(a) \textit{polynomial} \quad : \quad [\text{Double}] \rightarrow \text{Double} \rightarrow \text{Double}

such that for coefficients \( c_0, c_1, c_2, \ldots, c_n \) and any \( x \) the following holds:

\[
\text{polynomial} [c_0, c_1, c_2, \ldots, c_n] x = c_0 + c_1 \cdot x + c_2 \cdot x^2 + \cdots + c_n \cdot x^n
\]

\textbf{e.g.:} \quad \text{polynomial} [3,4,5] 100.0 = 50403.0

\textbf{Hint:} Use Horner’s rule:

\[
c_0 + c_1 \cdot x + c_2 \cdot x^2 + \cdots + c_n \cdot x^n = c_0 + x \cdot (c_1 + x \cdot (c_2 + \cdots + x \cdot (c_n \cdot \cdots)))
\]

\textbf{Solution Hints}
\begin{align*}
\text{polynomial} \quad &\quad : \quad [\text{Double}] \rightarrow \text{Double} \rightarrow \text{Double} \\
\text{polynomial} \quad &\quad [] \quad x = 0 \\
\text{polynomial} \quad &\quad (c : cs) \quad x = c + x \ast \text{polynomial} \quad cs \quad x
\end{align*}

\[
\text{polynomial1} \quad cs \quad x = \text{foldr} \ clandestine (\lambda \quad c \quad r \rightarrow \quad c + x \ast \quad r) \quad 0 \quad cs
\]

\[
\text{polynomial2} \quad cs \quad x = \text{foldr} \ clandestine (\lambda \quad c \rightarrow \quad (c +). (x \ast)) \quad 0 \quad cs
\]

\[
\text{polynomial3} \quad cs \quad x = \text{foldr} \ clandestine ((. (x \ast)) \cdot (+)) \quad 0 \quad cs
\]

If we swap the argument order, we can easily abstract away \( cs \). The “\( \lambda \)-lifting” of the argument to \texttt{foldr} however leads to rather unreadable code, presented here as a puzzle: Do the transformations leading there yourself!

\begin{align*}
\text{polynomial4} \quad &\quad : \quad \text{Double} \rightarrow [\text{Double}] \rightarrow \text{Double} \\
\text{polynomial4} \quad x \quad = \quad \text{foldr} \ clandestine ((. (x \ast)) \cdot (+)) \quad 0
\end{align*}

(b) \textit{findJump} \quad : \quad \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}

\textit{findJump} \quad d \quad [\quad x \quad] \quad = \quad \text{error} \quad "\textit{findJump: empty list}" \quad \textit{findJump} \quad d \quad [\quad x \quad] \quad = \quad \text{error} \quad "\textit{findJump: singleton list}"
**findJump** \( d \) \((x : xs \equiv (y : ys))\) = \( \text{if } \text{abs} (x - y) > d \)
\[\text{then} \ (x, y) \]
\[\text{else} \ \text{findJump} \ d \ xs\]

(c) \textit{suffixes} :: [a] \rightarrow [[a]]

delivers for each argument list all its suffixes, e.g.:

\textit{suffixes} \[1,2,3,4\] = [[1,2,3,4], [2,3,4], [3,4], [4], []]

(The order is irrelevant.)

**Solution Hints**

\textit{suffixes} :: [a] \rightarrow [[a]]

\textit{suffixes} [] = [[]]

\textit{suffixes} xs \equiv (y : ys) = xs : \textit{suffixes} ys

(d) \textit{diagonal} :: [[a]] \rightarrow [a]

interprets its argument as a matrix (represented as in Exercise 2.1), which may be assumed to be square, and returns the main diagonal of that matrix, e.g.:

\textit{diagonal} [[1,2,3], [4,5,6], [7,8,9]] = [1,5,9]

**Solution Hints**

\textit{diagonal}, \textit{diagonal'} :: [[a]] \rightarrow [a]

\textit{diagonal} [] = []

\textit{diagonal} ([]) : xss) = \text{error} "not square"

\textit{diagonal} ((x : xs) : xss) = x : \textit{diagonal} (\text{map tail} xss)

\textit{diagonal'} = \text{zipWith} ((\text{head}.) \circ \text{drop}) [0..]

Discuss the use of \textit{head} in the variant \textit{diagonal'}!

(e) \textit{isSquare} :: [[a]] \rightarrow \text{Bool}

determines whether its argument corresponds to a list-of-lists representation (as in Exercise 2.1) of a \textit{square} matrix.

**Solution Hints**

The following works only for finite lists of finite lists:

\textit{isSquare}, \textit{isSquare'} :: [[a]] \rightarrow \text{Bool}

\textit{isSquare} xs = \text{all} (((\text{length} \ xs) \equiv) \circ \text{length}) \ xs

\textit{isSquare'} xs = \text{all} (((\text{length} \ xs) \equiv) (\text{map} \ \text{length} \ xs)

(It is undecidable whether an infinite list of lists has only infinite element lists.)
Exercise 4.5 — Haskell Evaluation (30% of Midterm 1, 2003)

Assume the following Haskell definitions to be given:

\[
\begin{align*}
\text{foldr} & : (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\
\text{foldr} \ f \ e \ [\ ] & = e \\
\text{foldr} \ f \ e \ (x:xs) & = f \ x \ (\text{foldr} \ f \ e \ xs) \\
\text{concat} & = \text{foldr} \ (++) \ [\ ] \\
(\|\|) & : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} -- \text{Boolean disjunction: or} \\
\text{True} \ || \ _ & = \text{True} \\
\text{False} \ || \ b & = b
\end{align*}
\]

\[
\begin{align*}
\text{any} \ p & = \text{foldr} \ ((\|\|) \ . \ p) \ False \\
\text{gen} \ f \ (x,s) & = x : \text{gen} \ f \ (f \ x \ s) \\
\text{foo} \ k \ n & = (k + n, n + 2)
\end{align*}
\]

Simulate Haskell evaluation for the following expressions (write down the sequence of intermediate expressions):

(a) \(\text{foldr} \ (*) \ 1 \ [6,7]\)

(b) \(\text{any} \ (> \ 0) \ (\text{gen} \ \text{foo} \ (0,1))\)

Solution Hints

\[
\begin{align*}
\text{foldr} \ (*) \ 1 \ [6,7] \\
& = 6 \times (\text{foldr} \ (*) \ 1 \ [7]) \\
& = 6 \times (7 \times (\text{foldr} \ (*) \ 1 \ [\ ])) \\
& = 6 \times (7 \times 1) \\
& = 6 \times 7 -- X \\
& = 42
\end{align*}
\]

\[
\begin{align*}
\text{any} \ (> \ 0) \ (\text{gen} \ \text{foo} \ (0,1)) \\
& = \text{foldr} \ ((\|\|) \ . \ (> \ 0)) \ False \ (\text{gen} \ \text{foo} \ (0,1)) \\
& = \text{foldr} \ ((\|\|) \ . \ (> \ 0)) \ False \ (0 : \text{gen} \ \text{foo} \ (\text{foo} \ 0 \ 1)) \\
& = ((\|\|) . (> 0)) \ 0 \ (\text{foldr} \ ((\|\|) \ . \ (> 0)) \ False \ (\text{gen} \ \text{foo} \ (\text{foo} \ 0 \ 1))) \\
& = ((\|\|) . (> 0)) \ 0 \ (\text{foldr} \ ((\|\|) \ . \ (> 0)) \ False \ (\text{gen} \ \text{foo} \ (\text{foo} \ 0 \ 1))) \\
& = ((\|\|) . (> 0)) \ False \ (\text{gen} \ \text{foo} \ (\text{foo} \ 0 \ 1)) -- X \\
& = True (\text{foldr} \ ((\|\|) \ . \ (> 0)) \ False \ (\text{gen} \ \text{foo} \ (\text{foo} \ 0 \ 1))) \\
& = \text{foldr} \ ((\|\|) \ . \ (> 0)) \ False \ (\text{gen} \ \text{foo} \ (\text{foo} \ 0 \ 1)) \\
& = \text{foldr} \ ((\|\|) \ . \ (> 0)) \ False \ (\text{gen} \ \text{foo} \ (0 + 1, 1 + 2)) \\
& = \text{foldr} \ ((\|\|) \ . \ (> 0)) \ False \ ((0 + 1) : \text{gen} \ \text{foo} \ (\text{foo} \ (0 + 1) \ (1 + 2))) \\
& = ((\|\|) . (> 0)) \ (0 + 1) \ (\text{foldr} \ ((\|\|) \ . \ (> 0)) \ False \ (\text{gen} \ \text{foo} \ (\text{foo} \ (0 + 1) \ (1 + 2)))) \\
& = ((\|\|) . (> 0)) \ (0 + 1) \ (\text{foldr} \ ((\|\|) \ . \ (> 0)) \ False \ (\text{gen} \ \text{foo} \ (\text{foo} \ (0 + 1) \ (1 + 2)))) -- X \\
& = True (\text{foldr} \ ((\|\|) \ . \ (> 0)) \ False \ (\text{gen} \ \text{foo} \ (\text{foo} \ 1 \ (1 + 2)))) \\
& = True
\end{align*}
\]
Exercise 4.6 — Defining Haskell Functions  (20% of Midterm 1, 2004)

Define the following Haskell functions (the solutions are independent of each other):

(a) \textit{sum} :: \texttt{[Integer]} \rightarrow \texttt{Integer}

such that \textit{sum} \, \textit{xs} evaluates to the sum of all elements of the list \textit{xs}.

(b) \textit{all} :: (\texttt{a} \rightarrow \texttt{Bool}) \rightarrow \texttt{[a]} \rightarrow \texttt{Bool}

such that \textit{all} \, \textit{p} \, \textit{xs} evaluates to \texttt{True} if \textit{p} considered as a predicate holds for all elements of \textit{x},
and to \texttt{False} if there is at least one element in \textit{xs} for which \textit{p} does not hold.

E.g., \texttt{all \ (>1) [2..10] = True}

(c) \textit{selMod} :: \texttt{Integer} \rightarrow \texttt{[Integer]} \rightarrow \texttt{[Integer]}

such that \textit{selMod} \, \textit{k} \, \textit{xs} selects from the list \textit{xs} all those elements that are equivalent to \textit{k} modulo \textit{k} + 1, e.g.,

\texttt{selMod 2 [2, 3, 8, 1, 2, 5] = [2, 8, 2, 5]}

(d) \textit{sources} :: \texttt{Eq a} \Rightarrow \texttt{[(a,a)]} \rightarrow \texttt{[a]}

such that \textit{sources} \, \textit{ps} returns the \textit{sources} of the graph \textit{ps}.

Here, the list \textit{ps} of pairs is considered as representing a simple graph by representing each edge from node \textit{x} to node \textit{y} by the pair \((x,y)\).

The \textit{context} “\texttt{Eq a ⇒}” just means that you may use the equality test for elements of type \textit{a},
i.e., \((\Rightarrow) :: \texttt{a} \rightarrow \texttt{a} \rightarrow \texttt{Bool}.

Example: \texttt{sources \ [(2,3), (3,4), (1,4), (1,5), (2,5)] = [2,1]}

(The order is irrelevant.)

Solution Hints

\texttt{sum = foldl (+) 0}

\texttt{sum = foldr (+) 0}

\texttt{sum [ ] = 0}

\texttt{sum (x:xs) = x + sum xs}

\texttt{all = foldr (&&) True}

\texttt{all p [ ] = True}

\texttt{all p (x:xs) = p x && all p xs}

\texttt{selMod :: Integer \rightarrow [Integer] \rightarrow [Integer]}

\texttt{selMod k xs = [ x | x \leftarrow xs , x \ 'mod' \ (k+1) \equiv k ]}

\texttt{sources, sources' :: Eq a ⇒ [(a,a)] \rightarrow [a]}
\[
\text{sources } ps = \textbf{let} \ ( \text{srcs, trgs}) = \text{unzip } ps \\
\quad \textbf{in} \ \text{filter} \ (\text{'notElem' trgs}) \ srcs \\
\]

\[
\text{sources' } ps = \textbf{let} \ trgs = \ [ \ \text{snd} \ p \ | \ p \leftarrow \ ps \ ] \\
\quad \textbf{in} \ = \ [ \ x \ | \ (x, y) \leftarrow \ ps, \ x \text{'notElem' trgs} \ ]
\]