A crash course to Haskell

1 How to get it?

Haskell is freely available to install on all major platforms, including Window, Linux, and Mac OS X. We recommend to install the whole Haskell platform, which contains, among others:

- The Glasgow Haskell Compiler
- The Stack tool for developing projects
- Core packages and widely used packages

Go to `https://www.haskell.org/downloads/#platform` and follow the link for your platform (Linux/OS X/Windows) to download and install the Haskell components for your operating system.

The installation steps on Windows are:

1. Make sure you have PowerShell v2+ and and .NET Framework 4+ already installed
2. Start PowerShell as administrator via the Run command window:
   - Press Win Key + R. A a small window will pop up.
   - Type in **powershell** and press Ctrl+Shift+Enter or press and hold Ctrl+Shift.
   - Click OK to make PowerShell run as administrator
3. Install Chocolatey by running the command below in **powershell**:

   ```
   > Set-ExecutionPolicy Bypass -Scope Process -Force;
   [System.Net.ServicePointManager]::SecurityProtocol =
   [System.Net.ServicePointManager]::SecurityProtocol -bor 3072;
   iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))
   ```

   and wait a few seconds for the command to complete.
4. Run

   ```
   > choco install haskell-dev
   ```

You may be asked to confirm many installation steps by pressing `Y`

5. Run

```
> refreshenv
```

You may need to restart **powershell**, before being able ro run the command

```
> ghci
```

to run the GHC interactive environment.

2 How to work with Haskell?

We will use the GHCi to initiate Haskell sessions, load and execute programs.

GHCi stands for *GHC interactive environment*. It is a command-line tool that can be started by typing `ghci` at the command prompt:

```
> ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Prelude>
```

GHCi works in the **R**ead-**E**val-**P**rint (REPL) mode: it performs repeatedly the following sequence of operations:

1. It waits for us to type a definition, expression, or command. After we submit it (usually, by pressing `Enter`), the input is **read** and **eval**uated.
2. The result of evaluating an expression is **print**ed.

   The evaluation of a definition does not produce output (therefore nothing is printed), but it adds a binding to the environment.

   GHCi commands have side effects too (see subsection below).
3. The input prompt is shown again, to indicate that GHCi is waiting for another input.

Example:

```
> x = 3   -- a definition that binds x to 2; nothing to print
> x^2+2   -- compute the value of  x^2 + 2; print 11
11
```

Haskell recognizes comments: they start with '`--`' and extend to the end of line. Comments are used to document the written code, and are ignored (not read) by GHCi.

Documentation about how GHCi works is available online: click here

3 Haskell programs

A Haskell program is a text file consisting of a group of definitions grouped in a module.

- We can use any text editor to write Haskell programs. A Haskell program for a module *Name* should be named *Name*`.hs`. This is similar to the naming convention in Java: the source file of a class C should be named C`.java`
- The simplest structure of a source file for a module *Name* is

  `module` *Name* `where`
  $definition_1$
  `...`
  $definition_n$

Definitions are of the following kinds:

- Definitions which name expressions of different types, including functions. Examples

  ```
  x::Integer  -- if omitted, the type declaration is computed gy GHCi
  x = 1

  intsFrom::[Integer]->[Integer]
  intsFrom x = x:intsFrom (x+1)

  nats = intsFrom 1 -- GHCi will infer that nats::[Integer]
  ```

- Definitions of type classes, algebraic types, and type aliases (Lecture 6).

4 GHCi commands

The most important GHCi commands are:

`:load` *Name* reads definitions of module *Name* from file *Name*`.hs`

`:quit` quits the current working session with Haskell.

`:type` *expr* prints the type of *expr* without evaluating it.

`:set +t` sets GHCi to show the type of each variable bound by a statement. For example:

```
> (x:y:xs) = "abcdefgh"
x :: Char
xs :: [Char]
y :: Char
```

5 Useful predefined operations

For `Integer` arithmetic

`+`	The sum of two integers.
`*`	The product of two integers.
`^`	Raise to the power; `2^3` is 8.
`-`	The difference of two integers, when infix: `a-b`; the integer of opposite sign, when prefix: `-a`.
`div`	Whole number division; for example `div 14 3` is 4. This can also be written `14 ‘div‘ 3`.
`mod`	The remainder from whole number division; for example `mod 14 3` (or `14 ‘mod‘ 3`) is 2.
`abs`	The absolute value of an integer; remove the sign.
`negate`	The function to change the sign of an integer.

Relational operators

There are ordering and (in(equality relations over the integers, as there are over all basic types. These functions/operators take 2 integers as input and return a `Bol`, that is either `True` or `False`. The relations are

`>`	greater than (and not equal to)
`>=`	greater than or equal to
`==`	equal to
`/=`	not equal to
`<=`	less than or equal to
`<`	less than (and not equal to)

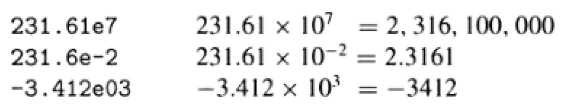
Floating-point numers: `Float`

Literal floats in Haskell can be given by decimal numerals, such as

```
3.141592
-34.25
567.123
16.0
```

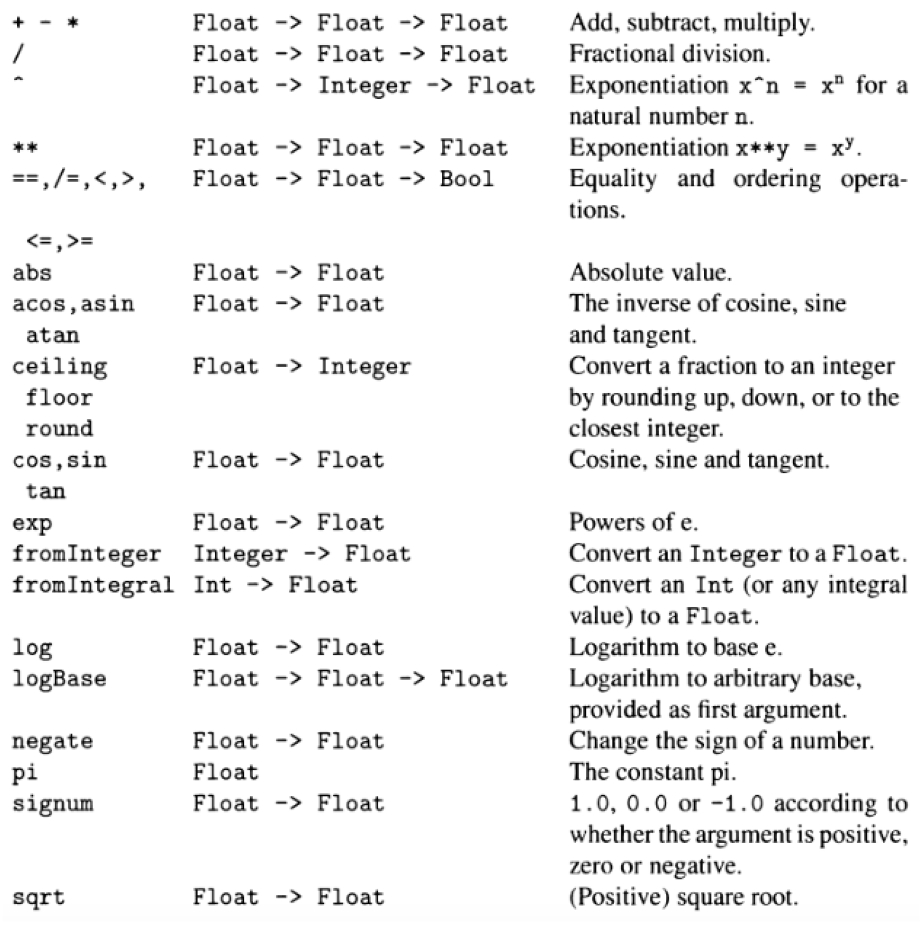
The numbers are called floating point because the position of the decimal point is not the same for all `Floats`; depending upon a particular number, more of the space can be used to store the integer or the fractional part.

Haskell also allows literal floating-point numbers in scientific notation. These take the form below, where their values are given in the right-hand column of the table:



`231.61e7`	$231.61 \times 10^7 = 2,316,100,000$
`231.6e-2`	$231.61 \times 10^{-2} = 2.3161$
`-3.412e03`	$-3.412 \times 10^3 = -3412$

This notation is more convenient than the decimal notation for very large and small numbers. Haskell provides a range of operators and functions over `Float`:



`+ - *`	`Float -> Float -> Float`	Add, subtract, multiply.
`/`	`Float -> Float -> Float`	Fractional division.
`^`	`Float -> Integer -> Float`	Exponentiation `x^n = x`^{n} for a natural number `n`.
`**`	`Float -> Float -> Float`	Exponentiation `x**y = x`^{y}.
`==,/=,<,>,` `<=,>=`	`Float -> Float -> Bool`	Equality and ordering operations.
`abs`	`Float -> Float`	Absolute value.
`acos,asin` `atan`	`Float -> Float`	The inverse of cosine, sine and tangent.
`ceiling` `floor` `round`	`Float -> Integer`	Convert a fraction to an integer by rounding up, down, or to the closest integer.
`cos,sin` `tan`	`Float -> Float`	Cosine, sine and tangent.
`exp`	`Float -> Float`	Powers of e.
`fromInteger`	`Integer -> Float`	Convert an `Integer` to a `Float`.
`fromIntegral`	`Int -> Float`	Convert an `Int` (or any integral value) to a `Float`.
`log`	`Float -> Float`	Logarithm to base e.
`logBase`	`Float -> Float -> Float`	Logarithm to arbitrary base, provided as first argument.
`negate`	`Float -> Float`	Change the sign of a number.
`pi`	`Float`	The constant pi.
`signum`	`Float -> Float`	`1.0`, `0.0` or `-1.0` according to whether the argument is positive, zero or negative.
`sqrt`	`Float -> Float`	(Positive) square root.

6 Getting started

The goal of Lab 5 is to illustrate a new style of functional programming, called **lazy programming**. This programming style is supported by languages that implement the **lazy evaluation** strategy, or its optimized version called **call-by-need** evaluation. Haskell is a language for functional programming which implements the call-by-need evaluation strategy.

In **lazy evaluation**, the arguments of a function call are evaluated only if their value is needed to compute the overall result. Moreover, if an argument is structured (a list or a tuple), only those parts of the argument which are needed will be computed.

Lazy evaluation has consequences on the style of programs we can write. Since an intermediate list will only be generated **on demand**, using partially generated intermediate list can reduce significantly the overall cost of computation We will illustrate this programming style with a series of examples.

6.1 Lazy programming

Our first Haskell program is a text file `HLab1.hs` with the following content:

```
module HLab1 where

intsFrom::Integer->[Integer]    -- type declaration
intsFrom x = x:intsFrom (x+1)   -- definition

nats::[Integer]                 -- type declaration
nats = intsFrom 1               -- definition
```

(During the lab, we will extend this program with new definitions.) This program contains two definitions, for variables `intsFrom` and `nats`. Definitions are preceded by type declarations of the form

name`::`*type*

with the intended reading "*name* has type *type*". Often, Haskell can compute the types of the variables that get defined. If this is the case, we can omit the type declaration. For example, the `ghci` command

```
> :type intsFrom 1
intsFrom 1 :: [Integer]
```

indicates that Haskell can compute the type of `intsFrom 1`, which is `[Integer]`. This computation is called **type inference**, and does not evaluate the expression `intsFrom 1`.

In lazy programming (including Haskell), the evaluation of a definition

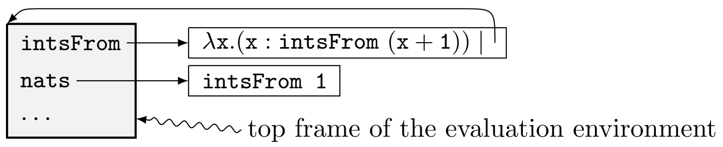
name `=` *expr*

adds a binding of *name* to *expr* to the top frame of the evaluation environment. This is different to what happens in strict functional programming (including Racket), where *name* is bound to the value of *expr*.

For example, the evaluation of the definitions

```
intsFrom x = x:intsFrom (x+1)
nats       = intsFrom 1
```

extends the top frame of the evaluation environment with two bindings, as illustrated below:


intsFrom
nats
...
λx.(x : intsFrom (x + 1))
intsFrom 1
top frame of the evaluation environment

In strict functional programming, the previous two definitions would be written as

```
(define (intsFrom x) (cons x (intsFrom (+ x 1))))
(define nats (intsFrom 1))
```

The first definition will add the same binding to the top frame, but the second definition will crash the system (stack overflow) because Racket will try to bind `nats` to the value `(intsFrom 1)`, and the evaluation of `(intsFrom 1)` is nonterminating and does not display anything:

$\underline{\texttt{(intsFrom 1)}} \rightarrow \texttt{(cons (1 }\underline{\texttt{(intsFrom 2)}}\texttt{)} \rightarrow \texttt{...}$

- `intsFrom` is bound to a function.

  `intsFrom x` computes the infinite list $[\mathtt{x}, \mathtt{x}+1, \mathtt{x}+2, \ldots]$
- `nats` is bound to an expression whose evaluation produces the infinite list of all integers starting from 1: `[1,2,3,...]`

If we ask Haskell to compute the value of `intsFrom 1`, it performs the same infinite computation like Racket:

$\underline{\texttt{intsFrom 1}} \rightarrow \texttt{1:}\underline{\texttt{(intsFrom 2)}}\ 1:2:\underline{\texttt{(intsFrom 3)}} \rightarrow \texttt{...}$

but Haskell will display the elements of the list as soon as they are generated:

```
> intsFrom 1
[1,2,3,4,...
```

The computation can be interrupted by pressing `Ctrl-C`.

`nats` is an example of defining an infinite data structure in lazy programming. The attempt to compute its complete value will run forever, but we can still use it in computations that need only finite portions of the structure rather than the whole value. The following examples illustrate such computations.

Getting the first n elements of an infinite list

Haskell has the built-in function `take n lst` which takes as inputs an integer $\mathtt{n} \geq 1$ and a list `lst`, and returns the list of first `n` elements of `lst`. If `lst` has less then `n` elements, then `take n lst` returns `lst`. The function `take` is easy to define by recursion on `n`:

```
take n _
  | (n <= 0)  = []              -- taking n ≤0 elements from any list yields []
take _ []     = []
take n (x:xs) = x:take (n-1) xs
```

To take the first 3 elements from `nats`, we must compute only the first 3 elements of `intsFrom 1`. This is what call-by-need evaluation does:

```
> take 3 nats
[1,2,3]
```

because

```
take 3 nats = take 3 (intsFrom 1)        -- need an element
  → take 3 (1:intsFrom 2)                -- get an element
  → 1:take 2 (intFrom 2)                 -- need an element
  → 1:take 2 (2:intsFrom 3)              -- get an element
  → 1:2:take 1 (intsFrom 3)              -- need an element
  → 1:2:take 1 (3:intsFrom 4)            -- get an element
  → 1:2:3:take 0 (intsFrom 4)            -- stop
  → 1:2:3:[] = [1,2,3]
```

Quiz

Consider the following definitions:

```
sieve1,sieveAll:[Integer]->[Integer]
sieve1 (x:xs) = x:filter (\y->(mod y x) > 0) xs
sieveAll (x:xs) = x:sieveAll (filter (\y->(mod y x) > 0) xs)
```

With list comprehensions (see Lecture Notes 5), the previous definitions can be rewritten in the more readable but equivalent form

```
sieve1 (x:xs) = x:[y | y<-xs, y `mod` x > 0]
sieveAll (x:xs) = x:sieveAll [y | y<-xs, y `mod` x > 0]
```

1. What does `sieve1 [n..]` compute for $\mathtt{n} \in \mathbb{N}, \mathtt{n} > 1$?

   Suggestion: check the results returned by `take 10 (sieve1 [n..])` for $\mathtt{n} \in \{2, 3, 4\}$

2. What does `sieve1 [1..]` compute? Does the computation terminate?

3. What does `sieveAll [2..]` compute?

Suggestion: add these definition to program `HLab1.hs`, reload it with command

```
:load HLab1
```

and test them.

1. `sieve1 [n..]` computes the infinite list of `Integer`s from `n`, without the multiples of `n` which are different from `n`. It is a nonterminating computation, For example:

   ```
   > sieve1 [2..]
   [2,3,4,5,6,...
   ```

2. `sieve1 [1..]` is a nonterminating computation of a list. It does not display anything because it runs indefinitely trying to find and display a nonexisting `n` $>$ `1` such that `mod n 1 > 0`. Only the first element, which is `1`, gets computed.

3. `sieveAll [2..]` computes the infinite list of all prime numbers:

   ```
   > sieveAll [2..]
   [2,3,5,7,11,13,17,19,23,...
   ```

Thus, we can define

```
primes = sieveAll [2..]
```

and use it to get finite portions of prime numbers. For example

```
take 2 primes
```

generates and gets only the first two prime numbers, because that is all we need:

```
take 2 primes = take 2 (sieveAll [2..])              -- need an element
 → take 1 2:sieveAll [y|y<-[3..],y `mod` 2 > 0]  -- get an element
 → 2:take 1 sieveAll [y|y<-[3..],y `mod` 2 > 0]  -- need an element
 → 2:take 1 3:sieveAll [y|y<-[4..],...]           -- get an element
 → 2:3:take 0 sieveAll [y|y<-[4..,...]]          -- stop
 → 2:3:[] = [2,3]
```

6.2 Proposed exercises

The following exercises will be discussed during the next lab.

1. Define recursively the function `map2::(a->b->c)->[a]->[b]->[c]`

   such that, if

   - `f` is a binary function that takes inputs of types `a` and `b`, and computes result of type `c`,
   - `lst1=[`$a_1,\ldots,a_n$`]` is a list of n elements of type `a`,
   - `lst2=[`$b_1,\ldots,b_n$`]` is a list of n elements of type `b`

   then `(map2 f lst1 lst2)` computes the list `[`$c_1,\ldots,c_n$`]` where c_i is the value of `(f` a_i b_i`)` for all $1 \leq i \leq n$. For example:

```
> map2 (+) [1,2,3] [4,5,6]        > map2 (*) [1,2,3] [4,5,6]
[5,7,9]                           [4.10,18]
```

2. Use `map` and `addLists` to define the infinite list of `Integers`

   `yList = [`$y_1, y_2, y_3, \ldots$`]`

   where $y_1 = y_2 = 1, y_3 = 2$ and $y_n = y_{n-1}^2 - 2 \cdot y_{n-2}^2 + 3 \cdot y_{n-3}$ for all $n > 3$.

3. Define the function `nestList::(Double->Double)->Double->[Double]` such that

   `nestList f v`

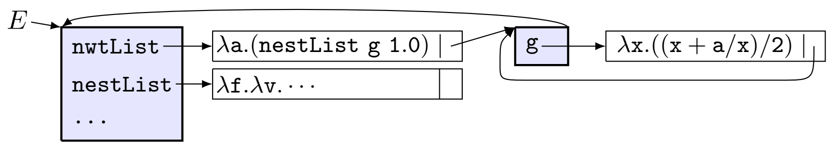
   computes the infinite list

   `[v, f v, f (f v),f (f (f v)),...]`

4. Consider the function `g` defined by

   ```
   nwtList a = let g x = (x+a/x)/2 -- same as g=\x.((x+a/x)/2)
               in nestList g 1.0
   ```

   This definition extends the top frame of the evaluation environment E with the binding for `nwtList`, as shown below:

   (a) What is the value of `nwtList 5.0`? Does the computation terminate?

   (b) What is the value of

   `head (take 5 nwtList 7.0)`

   Does the termination terminate?

   Suggestion: remember Newton's method to compute $\sqrt{x}$ when x is a positive real number.

5. Consider the function definition

   ```
   triples::Int->(Int,Int,Int)
   triples n = [(a,b,n-a-b)|a<-[1..n-2],b<-[1..n-1-a]]
   ```

   What is the value of the function call `(triples n)` when `n` > 2?

   Suggestion: use `ghci` to compute the values of `(triples n)` for some small values `n` > 2, to see what you get.

6. Define recursively the function `allTriples::Int->(Int,Int,Int)` such that, if `n` > 0, then `(allTriples n)` returns the list of all triples (a,b,c) with $a>0, b>0, c>0$ and $a+b+c\geq n$.

7. Define the list `t3List` of all triples (a,b,c) of type `(Int,Int,Int)` with $a>0, b>0, c>0$.

   Suggestion: Define recursively the recursive function

   ```
   triplesFrom n
   ```

   which generates the list of all triples (a,b,c) of type `(Int,Int,Int)` with $a>0, b>0, c>0$ and $a+b+c\geq n$.

8. A Pythagorean triple is a triple (a,b,c) of strict positive integers such that $a^2+b^2=c^2$. Define the function `(pythTriples n)` which returns the first `n` Pythagorean triples from `t3List`.

6.3 Miniproject 1

Power series have many applications in sciences and engineering. We consider power series of the form $\sum_{n=0}^{\infty} a_n\, x^n$ with $a_n \in \mathbb{R}$ for all $n\geq 0$, and represent them by infinite lists of `Double`s. For example

Ex. 1) $\displaystyle\sum_{n=0}^{\infty} x^n$ is represented by the list comprehension [1..]

Ex. 2) $\displaystyle e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$ can be represented by

```
eRepr = map (\x->1/fromInteger x) (coeffList [1..] 1 (*))
```

where

```
coeffList::[Integer]->a->(Integer->a->a)->[a]
-- coeffList [1..] x f computes the list [c1, c2, ...] where
-- c1 = x and cn = (f n cn-1) for all n > 1.
coeffList (n:ns) x f = x:coeffList ns (f n x) f
```

Ex. 3) $\displaystyle \sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots = \sum_{n=0}^{\infty} \frac{(-1)^n \cdot x^{2\,n+1}}{(2\,n+1)!}$ can be represented by

```
sinRepr = map c plist where
 c (x,y) = x/fromInteger y
 f n (_,y)
   | mod n 2 == 0 = (0,y*n)
   | mod n 4 == 1 = (1,y*n)
   | otherwise    = (-1,y*n)
 plist = coeffList [1..] (0,1) f
```

Note that `(coeffList [1..] (0,1) f)` generates on demand the infinite list of tuples

```
[(0,1),(1,1),(0,2!),(-1,3!),(0,4!),(1,5!),(0,6!),(-1,7!),...]
```

and `map c plist` generates on demand the list representation of $\sin x$. Another, less efficient, lazy implementation of the representation of e^x is the list comprehension

```
sinRepr = [(coeff n) | n<-[0..]] where
 coeff n = if (mod n 2) == 0
    then 0
    else ((-1)^(div (n+1) 2))/fromInteger (foldr (*) 1 [1..n])
```

Ex. 4) A polynomial $\sum_{k=0}^{n} a_k\, x^k$ is a power series too: $\sum_{k=0}^{\infty} a_k\, x^k$ where $a_i = 0$ for all $i > n$. A lazy implementation of the representation of the power series of the polynomial $\sum_{k=0}^{n} a_k\, x^k$ is

`[`$a_0, a_1 \ldots, a_n$`] ++ [0|<-i<-[1..]]`

Power series can be added, multiplied, scaled and divided as follows: if $r \in \mathbb{R}$, $a(x) = \sum_{n=0}^{\infty} a_n\, x^n$ and $b(x) = \sum_{n=0}^{\infty} b_n\, x_n$ are power series, then

$$a(x) + b(x) = \sum_{n=0}^{\infty} (a_n + b_n)\, x^n$$

$$r \cdot a(x) = \sum_{n=0}^{\infty} (r \cdot a_n)\, x^n$$

$$x \cdot a(x) = \sum_{n=0}^{\infty} a_n\, x^{n+1} = \sum_{n=0}^{\infty} c_n\, x^n \text{ where } c_n = \begin{cases} 0 & \text{if } n = 0, \\ a_{n-1} & \text{if } n > 0. \end{cases}$$

$$a(x) \cdot b(x) = \left(a_0 + x \cdot \sum_{n=0}^{\infty} a_{n+1}\, x^n \right) \cdot b = a_0 \cdot b(x) + x \cdot \left(b(x) \cdot \sum_{n=0}^{\infty} a_{n+1} x^n \right)$$

Also, if $b_0 \neq 0$ then

$$\begin{aligned} \frac{a(x)}{b(x)} &= \frac{a_0}{b_0} + x \cdot \frac{\sum_{n=1}^{\infty} (a_n - \frac{a_0}{b_0} \cdot b_n)\, x^{n-1}}{b(x)} \\ &= \frac{a_0}{b_0} + x \cdot \frac{\sum_{n=0}^{\infty} a_{n+1}\, x^n + (-\frac{a_0}{b_0} \cdot \sum_{n=0}^{\infty} b_{n+1}\, x^n)}{b(x)} \end{aligned}$$

Suppose `ra`, `rb` are representations of the power series $a(x) = \sum_{n=0}^{\infty} a_n\, x^n$ and $b(x) = \sum_{n=0}^{\infty} b_n\, x^n$. Define lazy implementations of the following functions:

```
eval::[Double] -> Double -> Int -> Double
sumS,prodS,divideS::[Double]->[Double]->[Double]
xprodS::[Double]->[Double]
rprodS::Double->[Double]->[Double]
```

such that

- `(eval ra v m)` computes $\sum_{n=0}^{\mathtt{m}} a_n\, \mathtt{v}^{\mathtt{n}}$
- `(sumS ra rb)` computes the representation of $a(x) + b(x)$
- `(rprodS r ra)` computes the representation of $r \cdot a(x)$ where r is the value of `r`
- `(xprodS a)` computes the representation of $x \cdot a(x)$
- `(prodS ra rb)` computes the representation of $a(x) \cdot b(x)$
- if $b_0 \neq 0$ then `(divS ra rb)` computes the representation of $\dfrac{a(x)}{b(x)}$

Suggestions

Note that

- The list a`:[0|i<=1..]]` represents the constant polynomial a.
- If `ra` represents the power series $\displaystyle\sum_{n=0}^{\infty} a_n\, x^n$ then

  `(0:ra)` represents the power series $\displaystyle\sum_{n=0}^{\infty} a_n\, x^{n+1}$

  `(tail ra)` represents the power series $\displaystyle\sum_{n=0}^{\infty} a_{n+1}\, x^n$
- You can make use of the function `map2` and of predefined functions `foldl`, `foldr`, `map`.

To test if your implementation is correct, try this:

```
> :{
ar,br,cr::[Double]
ar = [1,2,3]++[0|i<-[1..]]   -- 1 + 2 x + 3 x^2
br = 1:[0|i<-[1..]]          -- constant polynomial 1
cr = [1,-1]++[0|i<-[1..]]    -- 1 - x
:}
> take 4 (prodS ar cr)       -- (1 + 2 x + 3 x^2) · (1 - x) = 1 + x + x^2 - 3 x^3
[1.0,1.0,1.0,-3.0]
> take 6 (divides br cr)     -- 1/(1 - x) = 1 + x + x^2 + x^3 + ...
[1.0,1.0,1.0,1.0,1.0,1.0]
> eval sinRepr (pi/4) 16     -- compute an approximation of sin (pi/4)
0.7071067811865475
> eval eRepr 1 16            -- compute an approximation of e^1 = e
2.718281828458995
```