9. Lazy Evaluation

When passing **parameters** to a function, programming language design offers **two options** which are not mutually exclusive (both strategies can be implemented in the language):

- **applicative** (also called **strict**) evaluation strategy:
    - parameters are always evaluated **first**
    - can be further refined into: **call-by-value** (e.g. as it happens in *C*) and **call-by-reference** (e.g. as it happens for objects in *Java*).
- **normal** evaluation strategy (also called **non-strict**, and when implemented as part of the *PL* - **call-by-name**)
    - the function is always evaluated **first**
    - can be further refined into: **lazy**, which ensures that each expression is evaluated **at most once**

For more details, see the lecture on lazy evaluation. In *Haskell*, the default evaluation strategy is **lazy**.
There are ways in which we can force evaluation to be **strict**, however, in this lab, we will only explore several programming constructs which benefit from lazy evaluation.

9.0. Evaluation

9.0.0. Describe the **evaluation strategy** of the following expressions:

a.)

```
foldr ((||).(==1)) False [2,1,3,4,5,6,7]
```

b.)

```
foldl (flip ((||).(==1))) False [2,1,3,4,5,6,7]
```

9.1. Streams

There exists a explicit `Stream` type in Haskell, but for this lab we can think of **streams** as synonymous with **infinite** lists.
If you want to see their content, use the `take` function (`:t take`) to check the first n values.
9.1.1. Define the stream of **natural** numbers.

```
nats :: [Integer]
```

9.1.2. Using the stream of **natural** numbers, define the stream of **odd** numbers and **perfect squares**. Hint: use *higher order functions* for a quick solution.

```
-- 1, 3, 5, 7, ...
odds :: [Integer]


-- 0, 1, 4, 9, ...
squares :: [Integer]
```

9.1.3. Define the stream of **Fibonacci** numbers. Hint: `:t zipWith`.

```
fibs :: [Integer]
```

9.2. Infinite Binary Trees

You can use the following snippet to implement a `Show` instance for `BTree` that pretty prints the tree, you don't have to understand it, it should just format the Tree a bit nicer whenever it will get shown.
Click to hide ⇱

```
data BTree = Node Int BTree BTree | Nil


data PrintInfo = PrintInfo {
    len :: Int,
    center :: Int,
    text :: [String]
}


pp :: BTree -> PrintInfo
pp Nil = PrintInfo 3 2 ["Nil"]
pp (Node x l r) = seq check PrintInfo nlen ncenter ntext
  where
    check = if (length (show x)) > nlen then error "Nice try" else ()
    pp_l = pp l
    pp_r = pp r
    nlen = len pp_l + len pp_r + 1
    ncenter = len pp_l + 1
```

```haskell
    ntext = aligned_x : center_line : dotted_line : down_lines : combined_lines
      where
        aligned_x = replicate (ncenter - (div (length (show x)) 2) - 1) ' ' ++ show x
        center_line = replicate (ncenter - 1) ' ' ++ "|"
        dotted_line = replicate (center pp_l - 1) ' ' ++
                      replicate (nlen - center pp_l - center pp_r + 2) '-'
        down_lines = replicate (center pp_l - 1) ' ' ++ "|" ++
                     replicate (nlen - center pp_l - center pp_r) ' ' ++ "|"
        combined_lines = zipPad "" (combine) (text pp_l) (text pp_r)
          where
            zipPad :: a -> (a -> a -> a) -> [a] -> [a] -> [a]
            zipPad pad f [] [] = []
            zipPad pad f [] (y:ys) = y : zipPad pad f [pad] ys
            zipPad pad f (x:xs) [] = x : zipPad pad f xs [pad]
            zipPad pad f (x:xs) (y:ys) = f x y : zipPad pad f xs ys
            combine l r = l ++ replicate (len pp_l - length l + 1) ' ' ++ r

instance Show BTree where
    show = unlines . text . pp

tree :: BTree
tree = Node 1 (Node 2 Nil Nil) (Node 3 Nil Nil)
```

Defining a simple **binary tree** structure in *Haskell* is easy.
Take this for example:

```haskell
data BTree = Node Int BTree BTree | Nil deriving Show


{-      1
        |
   ---------
   |       |
   2       3
 -----   -----
 |   |   |   |
```

```
Nil Nil Nil Nil
-}


tree :: BTree
tree = Node 1 (Node 2 Nil Nil) (Node 3 Nil Nil)
```

But what if we want to enforce an **infinite binary tree**?
The solution is eliminating the need for the empty node `Nil`:

```
-- This is an infinite data type, no way to stop generating the tree
data StreamBTree = StreamNode Int StreamBTree StreamBTree


sbtree = StreamNode 1 sbtree sbtree
```

9.2.1. In order to view an **infinite tree**, we need to convert it to a **finite binary tree**. Define the function `sliceTree`, which takes a level `k`, an **infinite tree** and returns the first `k` levels of our tree, in the form of a **finite** one.

```
{-
> sliceTree 2 sbtree


       1
       |
   ---------
   |       |
   1       1
 -----   -----
 |   |   |   |
Nil Nil Nil Nil
-}


sliceTree :: Int -> StreamBTree -> BTree
```

9.2.2. Define the `repeatTree` function which takes an **Int** `k` and generates an **infinite tree**, where each node has the value `k`.

```
{-
> repeatTree 3
```

```
             3
             |
         ---------
         |       |
         3       3
      ------   ------
      |    |   |    |
      3    3   3    3
      .    .   .    .
      .    .   .    .
-}

repeatTree :: Int -> StreamBTree
```

9.2.3. Define the `generateTree` function, which takes a **root** `k`, a **left generator** function `leftF` and a **right generator** function `rightF`.
For example, let's say we have `k=2`, `leftF=(+1)`, `rightF=(*2)`. This should generate a tree where the *root* is 2 , the *left child* is $parent+1$
and the *right child* is $parent*2$.

```
{-
> generateTree 2 (+1) (*2)
                                     2
                                     |
               --------------------------------------------
               |                                          |
               3                                          4
        -----------------                          -----------------
        |               |                          |               |
        4               6                          5               8
    ---------       ---------                  ---------       ---------
    |       |       |       |                  |       |       |       |
    5       8       7       12                 6       10      9       16
  -----   -----    -----   -----             -----   -----    -----   -----
  |   |   |   |    |   |   |   |             |   |   |   |    |   |   |   |
```

```
  6   10  9   16   8   14  13  24         7   12  11  20   10  18  17  32
  .   .   .   .    .   .   .   .          .   .   .   .    .   .   .   .
  .   .   .   .    .   .   .   .          .   .   .   .    .   .   .   .
-}

generateTree :: Int -> (Int -> Int) -> (Int -> Int) -> StreamBTree
```

9.3. Numerical Approximations

9.3.1. Define the `build` function which takes a generator `g` and an initial value `a0` and generates the **stream**: `[a0, g a0, g (g a0), g (g (g a0)), ... ]`.

```
{-
    with this function, you should
    be able to easily define the natural numbers
    with something like: build (+1) 0
-}

build :: (Double -> Double) -> Double -> [Double]
```

9.3.2. Using the `build` function, define the following **streams**:

```
-- 0, 1, 0, 1, 0, 1, 0, ...
alternatingBinary :: [Double]

-- 0, -1, 2, -3, 4, -5, ...
alternatingCons :: [Double]

-- 1, -2, 4, -8, 16, ...
alternatingPowers :: [Double]
```

9.3.3. Define the `select` function which takes a **tolerance** `e` and a **stream** `s` and returns the `nth` element of the **stream** which satisfies the following condition: $|s_n - s_{n+1}| < e$.

```
select :: Double -> [Double] -> Double
```

Mathematical Constants

9.3.4. Knowing that $\lim_{n\to\infty}\frac{F_{n+1}}{F_n}=\varphi$, where F_n is the nth element of the **Fibonacci** sequence, write an approximation with **tolerance** `e=0.00001` of the **Golden Ration** (φ). Use the previously defined `fibs` **stream** and the `select` function.

```haskell
phiApprox :: Double
```

9.3.5. Consider the *sequence*:

$a_{n+1}=a_n+\sin(a_n)$; where a_0 is an *initial approximation*, randomly chosen (but **not** 0 because $a_{n+1}!=a_n$).
Knowing that $\lim_{n\to\infty}a_n=\pi$, write an approximation with **tolerance** `e=0.00001` of π . Make sure to use `build` and `select`.

```haskell
piApprox :: Double
```

Square Root

9.3.6. Given a number `k`, we want to create a function which calculates the **square root** of `k`. This is another place where **laziness** and **streams** come into play.

Consider the following *sequence*:

$a_{n+1}=\frac{1}{2}\left(a_n+\frac{k}{a_n}\right)$; where a_0 is an *initial approximation*, randomly chosen.
Knowing that $\lim_{n\to\infty}a_n=\sqrt{k}$, write a function that approximates $\sqrt{k}$ with **tolerance** `e=0.00001`. Use `build` and `select`.

```haskell
sqrtApprox :: Double -> Double
```

Derivatives

9.3.7. We can approximate the derivative of a function in a certain point using the definition of the derivative:

$f'(a)=\lim_{h\to 0}\frac{f(a+h)-f(a)}{h}$
We can obtain better successive approximations of the derivative in a point a , using a smaller h .
a) generate the sequence: $h_0,\frac{h_0}{2},\frac{h_0}{4},\frac{h_0}{8},\ldots$ (where h_0 is a randomly chosen *initial approximation*)
b) generate the list of approximations for $f'(a)$, using the formula above
c) write the function that takes a function f and a point a and approximates $f'(a)$ with tolerance `e=0.00001`, using the previous steps.

```haskell
derivativeApprox :: (Double -> Double) -> Double -> Double
```