Preparation and administration

- The second set of Haskell exercises comes in two parts: 2A and 2B. This is part 2A.
- Use the skeleton `Practicum2A.hs` from Canvas.
- Submit a single Haskell file based on `Practicum2A.hs` via Canvas.

Goal

The goal of the set 2A of Haskell exercises is to play with potentially infinite structures (connect with the evaluation strategies for untyped λ-calculus), to define Church numerals (connect with the definition of Church numerals in the untyped λ-calculus), and to define data types.

Lazy Evaluation

The method of computation in functional programming is the application of function to arguments. This is the same application as the application in the λ-calculus.

It may be the case that an expression admits different evaluations. Consider for example the function definition `double x = x + x`. The expression `double (double 2)` can be evaluated in different ways:

```
double (double 2)
= { unfold definition inner double }
double (2 + 2)
= {unfold definition double }
(2 + 2) + (2 + 2)
= {apply first +}
4 + (2+2)
= {apply last +}
4 + 4
= {apply +}
8
```

but also

```
double (double 2)
= { unfold definition outer double }
(double 2) + (double 2)
= {unfold definition left double }
(2 + 2) + (double 2)
= {apply first +}
4 + (double 2)
= {unfold definition double}
4 + (2+2)
= {apply second +}
4+ 4
= {apply +}
8
```

In this example, the order in which we apply the functions does not matter for the final result. This is a more general property of Haskell: for terminating expressions, the order in which functions are applied does not influence the final result. Recall that β-reduction in the untyped λ-calculus is confluent, and as a consequence every term has at most one normal form. Hence, if both the leftmost-outermost and the leftmost-innermost evaluation terminate, then they end in the same normal form (result).

In imperative programming the situation is different because of the possibilty of side effects. We have for example, with initially `n=0`:

```
n + (n:=1)
= { use (initial) value for n }
0 + (n:= 1)
= { perform assignment for n}
0 + 1
= { apply + }
1
```

whereas the following evaluation yields another result:

```
n + (n:=1)
= { perform assignment for n }
n + 1
= { use (new) value for n }
1 + 1
= { apply + }
2
```

Haskell uses the outermost, or call by need (sometimes also: call by name) evaluation strategy. In some cases, the outermost evaluation of an expression yields a result whereas the innermost evalualtion does not terminate. Consider for example the function definition `first (x,y) = x`. together with `inf = 1 + inf`. Clearly, the evaluation of `inf` does not terminate. The innermost evaluation of `first (0, inf)` does not terminate either. The outermost evaluation of `first (0,inf)` ends in `0`.

Recall that in the untyped λ-calculus with β-reduction, the leftmost-outermost strategy is normalizing, but the leftmost-innermost strategy is not.

Not all expressions in Haskell are evaluated in an outermost way. For example, the predefined functions `*` and `+` are evaluated in a so–called strict way, also called innermost or call–by–value. In addition, there is the possibility for the programmer to enforce a strict evaluation of a function.

Evaluation in Haskell has, besides being (mostly) outermost, two more important characteristics.

First, we do not evaluate below a lambda. For example, the expression `\x -> 1 + 2` is considered to be fully evaluated, even though the body of the lambda contains the redex `1+2`. The underlying idea is that functions are viewed as black boxes. This idea is modelled in the untyped λ-calculus by considering the weak head normal forms (WHNFs) as results, instead of the normal forms.

Second, if an expression is copied, as may happen in an outermost evaluation, then only one copy of the expression is kept, with in addition as pointers to it as required by the copying. This is called sharing. Sharing can also be modelled in (an extension of) the λ-calculus.

Summing up, the evaluation strategy of Haskell is outermost (with some exceptions), uses sharing, and does not select redexes under a lambda. It is called *lazy evaluation*. A basic intuition of lazy evaluation is that an argument of a function is only evaluated if it is needed in order to compute the final result.

One of the consequences of lazy evaluation is that we can work with potentially infinite structures, like for example infinite lists. A simple example of an infinite list is the following constant list consisting of only 1s:

```
ones = 1 : ones
```

Now we can for example ask for the first element in this infinite list:

```
head ones
```

The evaluation is roughly as follows:

```
head ones
  = { unfold ones }
head ( 1 : ones )
  = { apply head }
1
```

A strict language would require to first completely evaluate `ones` before `head` can be called. Haskell however only evaluates `ones` until it is clear what the first element is, and then calls `head`. We can also ask for more than only the first element:

```
take 100 ones
```

or also, if first we define

```
addFirstTwo (a:b:l) = a+b
```

then we can do for example

```
addFirstTwo ones
```

Haskell has predefined already for example the following:

```
[3 ..] = [3,4,5,6,...
[3,5 ..] = [3,5,7,...
```

Exercises infinite lists

Give recursive definitions, not enumerating definitions, as much as possible. For example, in the first exercise do not define `naturals` as `[1 ..]`.

1. Define the infinite list `naturals` $1, 2, 3, 4, 5, \ldots$.
2. Define the infinite list `zeroesandones` $0, 1, 0, 1, 0, 1, 0, 1, \ldots$.
3. Define the infinite list `threefolds` consisting of all numbers $3n$ for $n = 0, 1, 2, \ldots$.
4. Define the infinite list `nothreefolds` consisting of all positive integers, where all numbers $0 \times 3, 1 \times 3, 2 \times 3, \ldots$ have been removed. Use `'mod'`, and possibly also an auxiliary function `removeif` that takes as input a predicate and a list. and removes all elements from the list for which the predicate holds.
5. Define a function `allnfolds` that takes as input a number n and gives back as output the infinite lists of $0n, 1n, 2n, 3n, \ldots$.
6. Define a function `allnaturalsexceptnfolds` that takes as input a number n and gives back as output the infinite list consisting of all natural numbers where the numbers $0n, 1n, 2n, \ldots$ all have been removed.
7. Define a function `allelementsexceptnfolds` that takes as input a number n and a list l and gives back as output the list l where the numbers $0n, 1n, 2n, 3n, \ldots$ all have been removed.
8. Define the infinite list `eratosthenes` of prime numbers. Use the principle of the sieve of Eratosthenes:
   (a) We start with the infinite list $2, 3, 4, 5, 6, 7, \ldots$.
   (b) We take the first number p from the list; this is a prime number, and we remove all $0p, 1p, 2p, \ldots$ from the list.
   (c) We iterate step 2 with what remains of the list.
9. Define the infinite list `fibonacci` of Fibonacci numbers. Use the predefined function `zipWith`. It takes as input a function, a list, and another list, and gives back as output a list consisting of the function applied to both first elements, the function applied to both second elements, etcetera.

Church numerals

There are different ways to represent natural numbers in the λ-calculus. A well-known way is using the so-called *Church numerals*. The idea is roughly as follows: we are looking for infinitely many different closed normal forms that permit to do arithmetic. Further, the intuition is that we work with bound variables that stand for 'zero' and 'plus one'. See also the course notes on λ-calculus. In the practical work we are going to write the Church numerals in Haskell. This illustrates the relation between the λ-notation (on paper) and the abstraction in Haskell. Below we give the first four Church numbers in both the λ-notation and in the Haskell notation:

c_0	$\lambda s.\, \lambda z.\, z$	`\s -> \z -> z`
c_1	$\lambda s.\, \lambda z.\, s\, z$	`\s -> \z -> (s z)`
c_2	$\lambda s.\, \lambda z.\, s\, (s\, z)$	`\s -> \z -> (s (s z))`
c_3	$\lambda s.\, \lambda z.\, s\, (s\, (s\, z))$	`\s -> \z -> (s (s (s z)))`

Haskell functions are not printed. Here is the definition of `churchnumeral`:

```
churchnumeral n = if (n==0)
  then \s -> \z -> z
  else \s -> \z -> (churchnumeral (n-1) s (s z))
```

Haskell does not permit equality tests on functions. (Otherwise it would for example be able to solve the Collatz conjecture.) However, we can see whether two Church numerals are equal by translating them back to natural numbers. We define the following:

```
backtointeger n =
  n (\x -> x+1) 0
```

The idea is that `backtointeger` takes as input a Church numeral. That Church numeral is applied to the successor and to the zero. The output reveals the natural number that was represented by the Church numeral. (Note: it does not matter much whether we take take Church numerals with first abstraction over 'zero' and then abstraction over 'successor' or the other way around.) Example:

```
backtointeger (churchnumeral 3) ;;
```

indeed yields:

```
*Main> backtointeger (churchnumeral 3)
3
```

Exercises Church numerals

1. Use the functions as defined above.

   Try to use `churchnumeral` and `backtointeger`.

2. Define a function `churchequality` that tests equality of two Church numerals by translating them back to natural numbers. Use `backtointeger`.

   Test your function (you do not need to write down the tests in your file). We should have for example:

   ```
   *Main> churchequality (churchnumeral 3) (churchnumeral 3)
   True
   *Main> churchequality (churchnumeral 3) (churchnumeral 2)
   False
   ```

3. Define the function `successor`, using the first definition of successor on the Church numerals as explained in the course notes on λ-calculus.

   Test your function (you do not need to write down the tests in your file). We should have for example:

   ```
   *Main> backtointeger (successor (churchnumeral 3))
   4
   ```

4. Give also the second definition of successor (as explained in the course notes) in Haskell. Call it for example `successorb`.

   Test your function (you do not need to write down the tests in your file).

5. Define a function `apply1` that takes as input a unary function `f` on Church numerals and a natural number `n`, and that gives back as output the result of applying `f` to the Church numeral for `n`, but read back as natural number. Hence use `backtointeger`.

   We should for example have the following:

   ```
   *Main> apply1 successor 3
   4
   *Main> apply1 successorb 5
   6
   ```

6. Define the functions `addition`, `multiplication`, and `exponentiation` on Church numerals. (See the course notes for the definitions in λ-calculus).

7. Define the function `apply2` that takes as input a binary function on Church numerals `f`, and two natural numbers `n` and `m`, and gives back as output the result of applying `f` to the Church numerals for `n` and `m`, but read back as natural number. Hence use again the function `backtointeger`.

   We should for example have the following:

```
*Main> apply2 addition 3 5
8
*Main> apply2 multiplication 3 5
15
*Main> apply2 exponentiation 3 5
243
```

Binary trees

We define the following data type:

```
data Binarytree a = Leaf | Node (Binarytree a) a (Binarytree a)
  deriving Show
```

This permits to have for example

```
exampleTreeA = Leaf
exampleTreeB = Node Leaf 1 Leaf
```

The data type is used to represent binary trees with labels of type `a` on the nodes. The type `a` can be instantiated, for example by `Integer`. Therefore the type is *polymorphic*. Further, `deriving Show` permits us to use show and print on elements of type `Binarytree` on the screen. An example of a function:

```
single :: a -> Binarytree a
single x = Node (Leaf) x (Leaf)
```

First we explicitly give the type. Next we define a function that given an input `x` of type `a` creates a binary tree with one node labelled with `x`. Try the following:

```
*Main> single 5
Node Leaf 5 Leaf
```

Note that the compiler derives itself the instance of `a` that we use.

Exercises binary trees

1. Define a function `numberofnodes` that takes as input a `Binarytree` and gives back as output the number of nodes.
2. Define a function `height` that takes as inout a `Binarytree` and gives back as output the height of the tree (which is the maximal length of a path from the root to a leaf).
3. Define a function `sumnodes` that takes as input a `Binarytree` and gives back as output the sum of all labels of the tree (and 0 in case the input is a leaf).

4. Define a function `mirror` that takes as input a `Binarytree` and gives back as output the mirrorred variant of that tree. We should for example have:

   ```
   *Main> mirror (Node (single 5) 3 Leaf)
   Node Leaf 3 (Node Leaf 5 Leaf)
   ```

5. Define a function `flatten` that takes as input a `Binarytree` and gives back as output the list of labels from left to right, as obtained in an inorder traversal of the tree.

6. Define a function `treemap` that takes as input a function and a `Binarytree`, and gives back as output the function applied to all labels. This is analoguous to the function `map` for lists.

Exercises binary search trees

We now consider binary search trees, which are binary trees where the nodes are labelled as above, but with in addition the following property: the labels are totally ordered, and for every node n with label l the labels of all nodes m in the left sub-tree of n are smaller than l, and the labels of all nodes o in the right sub-tree of n are larger than l. We assume all labels to be different. This kind of binary search trees are sometimes used to implement sets.

1. Define a function

   ```
   smallerthan :: (Ord a) => a -> BinaryTree a -> Bool
   ```

   which takes as input an element of some ordered type `a`, and a binary tree with labels of type `a`, and returns `True` exactly if all labels in the input-tree are strictly smaller than the first input.

   Analoguously, define a function

   ```
   largerthan :: (Ord a) => a -> BinaryTree a -> Bool
   ```

   checking whether all labels of the input-tree are strictly larger than the first input.

2. Define a function

   ```
   isbinarysearchtree :: (Ord a) => BinaryTree a -> Bool
   ```

   that checks whether its input-tree is a binary search tree. You may wish to use `smallerthan` and `largerthan`.

3. Define a function

   ```
   iselement :: (Ord a, Eq a) => a -> BinaryTree a -> Bool
   ```

that takes as input an integer and a binary search tree, and gives back as output `True` exactly if the integer occurs as label in the tree.

4. Define a function

   ```
   insert :: (Ord a, Eq a) => a -> BinaryTree a -> BinaryTree a
   ```

   that takes as input an element of `a` and a binary search tree, and gives back as output the input-tree if the first input already occurs as a label in the input-tree, and otherwise gives back as output the tree obtained by adding the first input to the input-tree. The input-tree is assumed to be a binary search tree. The output-tree is required to be a binary search tree.

5. Define a function

   ```
   createbinarysearchtree :: (Ord a, Eq a) => [a] -> BinaryTree a
   ```

   that takes as input a list and gives back as output a binary search tree containing all elements of the input-list, having removed possible double occurrences of an integer in the input-list.

6. Define a function

   ```
   remove :: (Ord a, Eq a) => a -> BinaryTree a -> BinaryTree a
   ```

   that takes as input a label and a binary search tree, and gives back as output the input-tree in case it did not contain the input-label, and the input-tree from which the input-label is removed in the other case. Use an auxiliary function for removing the root from a binary search tree. Recall that a label occurs at most once in a binary search tree. (Do not do the following: flatten the input-tree, remove the to be removed element from the list, build a binary search tree fro the resulting list.)

Exercise Hanoi

This exercise is about the Towers of Hanoi. Define a function `hanoi` for the Towers of Hanoi. The 4 inputs of the function are the number of discs, and a representation of the first, second, and third rod. An option (not obligatory) is to use the following:

```
type Rod = String
type Move = (Integer, Rod, Rod)
hanoi :: Integer -> Rod -> Rod -> Rod -> [Move]
```

Define the function `hanoi` for the Towers of Hanoi. The first argument is the number of discs. The second, third, and fourth arguments represent the three rods. A move indicates which disc is moved, and from which rod to which rod. Explain in comments how your discs are numbered.

You can use an alternative approach instead. But in any case: write an elegant program and mention in comments sources in case you used some.