universität innsbruck

Functional Programming	WS 2020	LVA 703025
Exercise Sheet 1, 10 points		Deadline: Wednesday, October 14, 2020, 6am

- Please upload .hs-file(s) for exercises 1.1 and 1.3 in OLAT. You can upload separate files or merge the solutions into a single .hs-file.
- You can also put all your remaining solutions, e.g., for exercise 1.2 into some .hs-file (using Haskell-comments) or create a separate .txt-file and upload it.
- You can use a template .hs-file that is provided on the proseminar page.
- Each .hs file must compile with ghci.
- Don't forget to mark your completed exercises in OLAT.

Exercise 1.1 *Functions* **4 p.**

Below we have given four erroneous functions. Your task is to find the errors and fix them. The functions are provided in the Haskell file `template_01.hs`. Update the functions in this file and upload your solution to OLAT.

1. ```
   f :: Integer -> Integer
   f x y = (x + y) * 2
   ```
2. ```
   g :: Integer -> Integer -> Integer
   g x z = 2 * Z + x
   ```
3. ```
   h :: Integer -> Integer
   x h = x + 2
   ```
4. ```
   i :: integer -> integer
   i x = 2
   ```

Exercise 1.2 *Evaluation Strategies* **3 p.**

In the lecture on slide 2/14, it is shown that a single expression can be evaluated in different ways. For instance, consider the following expression:

```
(square 3) + (square (3 * 7))
```

This expression can be evaluated at three different positions:

1. one can start to evaluate `3 * 7` to `21`
2. one can start to evaluate `square 3` to `3 * 3`
3. one can start to evaluate `square (3 * 7)` to `(3 * 7) * (3 * 7)`

Note that one cannot yet evaluate the addition since for built-in operations like `+`, `*`, etc., all arguments must first be fully evaluated to normal form.
Each programming language fixes a certain evaluation strategy, that determines which position to start with. There are two common ones:

- An *innermost strategy* always evaluates the arguments before applying the function definition itself. Hence, an innermost strategy would start with steps (1) or (2), but not with (3), since there the argument `3 * 7` is not yet evaluated.

- An *outermost strategy* is the opposite. In the evaluation of a function application `f exp_1 ... exp_n` for some user-defined function `f`, none of the arguments `exp_i` is evaluated, but the function definition of `f` is applied. Only arguments of built-in operators like `+`, `*`, etc. must be evaluated. Hence, an outermost strategy would start with steps (2) or (3).

Now consider the following Haskell program.

```
square :: Integer -> Integer
square x = x * x

three :: Integer -> Integer
three x = 3

bot :: Integer -> Integer
bot x = bot (x + 1)
```

Your task is to perform step-by-step evaluations of the upcoming three expressions for both innermost- and outermost-strategy. For each expression, compare which strategy is more efficient.

1. `square (5 - 3)`
2. `three (5 - 3 * 2)`
3. `three (bot 1)`

Can you figure out which evaluation strategy Haskell is using, just by invoking one of these expressions within `ghci`?

Exercise 1.3 *Reusing functions* 3 p.

The previous exercise showed that the evaluation order determines the number of computation steps necessary to arrive at a result. In this exercise we show that also the function definitions in an Haskell program have an influence on the number of computations that will be executed.

In this exercise the goal is reuse functions such that the number of multiplications is minimized. Below we have given a function for calculating x^2 and two different implementations for calculating x^4:

```
square :: Integer -> Integer
square x = x * x

pow_4a :: Integer -> Integer
pow_4a x = x * x * x * x

pow_4b :: Integer -> Integer
pow_4b x = square (square x)
```

1. Show (using an innermost evaluation strategy) that the function `pow_4b` requires less multiplication steps than `pow_4a`.
2. Write a Haskell function `pow_16:: Integer -> Integer` for calculating x^{16} that uses the least amount of multiplication steps. It is **not** allowed to use the built-in exponentiation operator `^`. How many multiplications are necessary?

   Hint: it is possible to define `pow_16` without Haskell's multiplication operator `*`, but instead reuse the functions defined above.
3. Write a Haskell function `pow_20:: Integer -> Integer` for calculating x^{20} that uses as few multiplications as possible. Again, it is **not** allowed to use the built-in exponentiation operator `^`, but it is allowed to define auxiliary functions. How many multiplications are necessary?