[Due Date: May, 15, 2pm]

CS 381, Spring 2018 — Homework 3 (Types)

Please follow carefully ***all*** of the following steps:

1. Prepare a Haskell or literate Haskell file (ending in `.hs` or `.lhs`, respectively) that compiles without errors in GHCi. (Put all non-working parts and all non-code answers in comments.)
2. Submit ***only one*** solution per team (each team can have up to 5 members), through the COE TEACH web site. List the names of all team members as a comment in the file.
3. Hand in a ***printed*** copy of your solution before class on May, 15, or make otherwise sure that the TAs receive the copy before the deadline. Make sure that all lines are readable on the printout.

Late submissions will ***not*** be accepted. Do ***not*** send solutions by email.

Exercise 1. A Rank-Based Type Systems for the Stack Language

We extend the simple stack language from Homework 2 (Exercise 1) by the following three operations.

- `INC` increments the topmost element on the stack
- `SWAP` exchanges the two topmost elements on the stack, and
- `POP k` pops `k` elements of the stack.

The abstract syntax of this extended language is as follows.

```
type Prog = [Cmd]

data Cmd = LD Int
         | ADD
         | MULT
         | DUP
         | INC
         | SWAP
         | POP Int
```

Even though the stack carries only integers, a type system can be defined for this language that assigns *ranks* to stacks and operations and ensures that a program does not result in a rank mismatch.

The rank of a stack is given by the number of its elements. The rank of a single stack operation is given by a pair of numbers (n, m) where n indicates the number of elements the operation takes from the top of the stack and m is number of elements the operation puts onto the stack. The rank for a stack program is defined to be the rank of the stack that would be obtained if the program were run on an empty stack. A *rank error* occurs in a stack program when an operation with rank (n, m) is executed on a stack with rank $k < n$. In other words, a rank error indicates a stack underflow.

(a) Use the following types to represent stack and operation ranks.

```
type Rank    = Int
type CmdRank = (Int,Int)
```

First, define a function `rankC` that maps each stack operation to its rank.

```
rankC :: Cmd -> CmdRank
```

Then define a function `rankP` that computes the rank of a program. The `Maybe` data type is used to capture rank errors, that is, a program that contains a rank error should be mapped to `Nothing` whereas ranks of other programs are wrapped by the `Just` constructor.

```
rankP :: Prog -> Maybe Rank
```

Hint. You might need to define an auxiliary function `rank :: Prog -> Rank -> Maybe Rank` and define `rankP` using `rank`.

(b) Following the example of the function `evalStatTC` (defined in the file `TypeCheck.hs`), define a function `semStatTC` for evaluating stack programs that first calls the function `rankP` to check whether the stack program is type correct and evaluates the program only in that case. For performing the actual evaluation `semStatTC` calls the function `sem`.

Note that the function `sem` called by `semStatTC` can be simplified. Its type can be simplified and its definition. What is the new type of the function `sem` and why can the function definition be simplified to have this type? (You do not have to give the complete new definition of the function.)

Exercise 2. Shape Language

Recall the shape language defined in the slides on semantics. Here is the abstract syntax of the shape language (see also the file `Shape.hs`).

```
data Shape = X
           | TD Shape Shape
           | LR Shape Shape
           deriving Show
```

We define the type of a shape to be the pair of integers giving the width and height of its bounding box.

```
type BBox = (Int,Int)
```

A type can be understood as a characterization of values, summarizing a set of values at a higher level, abstracting away from some details and mapping value properties to a coarser description on the type level. In this sense, a bounding box can be considered as a type of shapes. The bounding box classifies shapes into different bounding box types. (The bounding box type information could be used to restrict, for example, the composition of shapes, such as applying TD only to shapes of the same width, although we won't pursue this idea any further in this exercise.)

(a) Define a type checker for the shape language as a Haskell function

```
bbox :: Shape -> BBox
```

(b) Rectangles are a subset of shapes and thus describe a more restricted set of types. By restricting the application of the TD and LR operations to rectangles only one could ensure that only convex shapes without holes can be constructed. Define a type checker for the shape language that assigns types only to rectangular shapes by defining a Haskell function

```
rect :: Shape -> Maybe BBox
```

Exercise 3. Parametric Polymorphism

(a) Consider the functions `f` and `g`, which are given by the following two function definitions.

```
f x y = if null x then [y] else x

g x  y = if not (null x) then [] else [y]
g [] y = []
```

(1) What are the types of `f` and `g`?
(2) Explain why the functions have these types.
(3) Which type is more general?
(4) Why do `f` and `g` have different types?

(b) Find a (simple) definition for a function `h` that has the following type.

```
h :: [b] -> [(a, b)] -> [b]
```

Note that the goal of this part of the exercise is *not* to find a particularly useful function, but a function definition that has the given type signature. More precisely, you should give a definition for `h` ***without*** a type annotation, for which GHCi will then infer the shown type. A perfectly valid way to approach this exercise is to experiment with function definitions, without giving type declarations, and have GHCi infer the type using the `:t` interpreter command.

(c) Find a (simple) definition for a function `k` that has the following type.

```
k :: (a -> b) -> ((a -> b) -> a) -> b
```

All remarks from part (b) apply here as well.

(d) Can you define a function of type `a -> b`? If yes, explain your definition. If not, explain why it is so difficult.