CS 381, Spring 2025 — Homework 4 (Types)

Important Reminders!

1. Upload your solution as a single Elm file (ending in `.elm`) to **Canvas**. (*Note*: No pdf; don't use Pensieve.)
2. **Only submit files that compile without errors!** (Put all non-working parts in comments.)
3. You must do all homework assignments by yourself, without the help of others. Also, you must not use services such as Chegg or Course Hero. If you need help, simply ask on Canvas, and we will help!
4. You can work in teams of ***up to four*** students to create and submit a common homework solution. To this end, first create an alphabetically sorted (by last name) list of the team members, and add this information at the beginning of the submitted Elm file, like so.

```
{- GROUP:
   Helena Eagen
   Dylan George
   Mark Scout
-}
```

The other group members **must not** submit a solution. All group members will receive the same grade.

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

We extend the simple stack language from Homework 3 (Exercise 1) by the following three operations.

- `DEC` decrements 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 Op = LD Int | ADD | MULT | DUP | DEC | SWAP | POP Int
type alias Prog = List Op
```

Even though the stack carries only integers, we can identify types for the stack operations that capture the constraints on the number of arguments they need on the stack. Specifically, we can define a type system that assigns *ranks* to stacks and operations, ensuring 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 stack operation is given by a pair of numbers (n, m) where n is the number of elements taken from the top of the stack and m is number of elements put 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 alias Rank   = Int
type alias OpRank = (Int,Int)
```

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

```
rankOp : Op -> OpRank
```

Then define a function `rankProg` 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.

```
rankProg : Prog -> Maybe Rank
```

Hint. It might be helpful to define an auxiliary function `rank : Prog -> Rank -> Maybe Rank` and define `rankProg` using `rank`.

(b) Following the example of the function `evalTC` (defined on slide 33 of the type system slides), define a function `semTC` for evaluating stack programs that first calls the function `rankProg` to check whether the stack program is type correct and evaluates the program only in that case. For performing the actual evaluation `semTC` calls the function `semProg` (as defined in the previous homework; see the file `HW3_Semantics.elm`).

However, the function `semProg` called by `semTC` can be simplified, especially, its type. What is the new type of `semProg`, and why is it safe to use this new type?

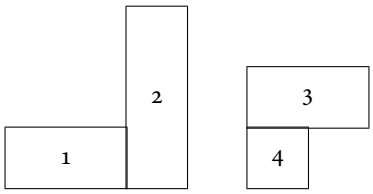
Exercise 2. Shape Language

Consider the following language for building shapes out of unit squares through horizontal and vertical composition.

$$s \in \mathit{shape} ::= \square \mid s;s \mid s \,\text{↱}\, s$$

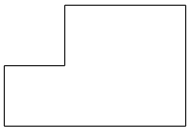
In this grammar, $\square$ denotes a unit square (or pixel) of width and height 1. The expression $s_1;s_2$ puts s_2 next to s_1 (on the right), which means to join s_2's left border with s_1's right border while aligning their bottom sides. The expression $s_3 \,\text{↱}\, s_4$ puts s_3 above s_4, which means to join s_3's lower border with s_4's upper border while aligning their left sides.

The borders of any shape are given by the smallest enclosing rectangle, which is also called the shape's *bounding box*; this means the lower border is given by the lowest contained square(s), the left border is given by the leftmost square(s), and so on. The effect of the operations is illustrated below.



Here, shape 4 is given by $\square$, the shapes 1 and 3 are given by $\square;\square$, and shape 2 is given by $\square \,\text{↱}\, (\square \,\text{↱}\, \square)$. The left shape is given, for example, by $(\square;\square);(\square \,\text{↱}\, (\square \,\text{↱}\, \square))$, and the right shape is given by $(\square;\square) \,\text{↱}\, \square$. We can observe that in many cases the same shape can be built in different ways. For example, shape 2 can also be given by $(\square \,\text{↱}\, \square) \,\text{↱}\, \square$.

The width and height of a shape's bounding box can be considered its type. For example, the type of shapes 1 and 3 are both $(2, 1)$, the type of shape 2 is $(1, 3)$, and the type of shape 4 is $(1, 1)$. The type of the following shape is $(3, 2)$.



The abstract syntax of the shape language is given by the following Elm type. The constructors `LR` ("left-right") and `TB` ("top-bottom") are the abstract syntax representations of `;` and ↱, respectively.

```
type Shape = X | LR Shape Shape | TB Shape Shape
```

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

```
type alias 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.

(a) Define a type checker for the shape language as an Elm function with the following type.

```
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 the following Elm function.

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