COM2001 - Spring Semester 2017–2018

Assignment 1: Boxes and Tokens

Deadline (MOLE): 3pm Wednesday 21st March (Week 7)

In this programming assignment you will implement a model of computation called the *Box-And-Token computer* (*BAT-computer*). You will not find any descriptions of it online, but it is a fully functional computer model that you can easily make at home using cardboard boxes (or boxes drawn on paper) and tokens (e.g., coins or cornflakes). The only other thing you need is a *program* (explained below).

To help you develop your code, there is a template file (`com2001ass1.hs`) on MOLE. Answer the questions below by editing a copy of that file as required. You should submit your final solution on MOLE as a single file (in `.hs` format).

DO NOT RENAME ANY CLASSES OR FUNCTIONS IN THIS ASSIGNMENT

After you submit your code, it will be tested using a set of test cases. *Your final grade for this assignment will depend in part on how many of those tests your code passes.* If you change any of the function or class names the tests will fail, and this may seriously reduce your overall grade. If you need additional information about standard classes and functions defined in the Haskell Prelude, you should consult the manuals at `http://www.haskell.org`. In particular, information about standard Haskell classes can be found at `https://www.haskell.org/tutorial/stdclasses.html`.

MARKING

This assignment is worth 50% of this semester's mark for the module (i.e. 25% of the entire module mark). The total number of marks available for this assignment is 40. Of these 40 marks, 30 will be determined automatically by checking your code against various test-cases using GHCi. The remaining 10 marks will be awarded for comprehensive and accurate documentation of your code.

PLAN AHEAD

We currently expect there to be two separate Software Hut (COM3420) deadlines in Week 7 as well. If this (or any other conflicting deadline) affects you, make sure you plan ahead and do the work sooner rather than later. *This is why this assignment has been handed out so early — it is to give you more flexibility when organising your own work schedule.* Don't carry out any last-minute edits to your code (e.g., adding your name in a comment as an afterthought) as this often leads to compilation errors. Make sure your code does what you want it to, and then avoid the temptation to change anything. *If your submitted code fails to load*, the test-cases will all fail and your mark will suffer accordingly.

1 The BAT-computer

For the purposes of this assignment, a computer is something that moves from one configuration to another as a result of following instructions in a *program*. It can be initialised by loading one or more inputs, and typically provides a single output if/when the program stops running. For our purposes all input and output values are of type `Int`.

```haskell
type Input  = Int
type Output = Int

class (Eq cfg) ⇒ ProgrammableComputer cfg where
  initialise   :: Program → [Input] → cfg
  getOutput    :: cfg → Output
  acceptState  :: Program → cfg → Bool
  doNextMove   :: Program → cfg → cfg
  runFrom      :: Program → cfg → cfg
  runProgram   :: Program → [Input] → cfg
  -- Default implementation
  runProgram p is = runFrom p (initialise p is)
```

In order to implement a computational model, we need to say what its configurations look like, and then implement each of the functions in this class.

A BAT-computer is a device comprising a finite list of *boxes*, $(\mathsf{Box}_0, \mathsf{Box}_1, \dots)$, each of which can contain finitely many *tokens*. The current configuration of a BAT-computer is given by saying how many tokens are in each box, and which instruction should be executed next. There are only three types of instruction. Two of these adjust the number of tokens in a box. The notation for each of these instructions, and its intended meaning, is as follows:

- `CLR x` ("clear x")
  Remove all of the tokens from Box_x;

- `INC x` ("increment x")
  Add one token to Box_x;

To run a program, you first load the user's input(s) into boxes Box_1, Box_2, Box_3, ... (Box_0 is reserved for use by the computer itself). Then you run the program, one instruction after another, unless instructed to do otherwise by a *jump* instruction. Jump instructions look like this:

- `JEQ x y n` ("jump-on-equal")
  If Box_x and Box_y contain the same number of tokens, run instruction I_n next;

After jumping to instruction n, you would then continue running instructions in sequence (so the next instruction after I_n would be I_{n+1}, unless I_n is itself another jump instruction).

```haskell
data Instruction
  = CLR {box :: Int}
  | INC {box :: Int}
  | JEQ {box1   :: Int,
         box2   :: Int,
         target :: Int}
  deriving (Eq, Show)

type Program = [Instruction]
```

Problem 1. Each instruction in a program refers to one or more boxes. The function `maxBoxNum` should identify how many boxes are used in the program as a whole, i.e. what is the highest value n for which Box_n is used by at least one instruction in the program? Implement this function.

```haskell
maxBoxNum :: Program → Int
maxBoxNum ...
```

As explained above, the configuration of a BAT-computer is given once you know how many tokens are in each box, and which instruction should be executed next (this information is given using a *program counter*).

```
data BATConfig = BATConfig {
    boxes   :: [Int],
    counter :: Int
    } deriving (Eq)
```

Problem 2. Make `BATConfig` an instance of the class `Show`, so that configurations look like this when displayed on the screen

```
boxes = <list of box values>; counter = <counter value>
```

For example, if the machine has just been initialised with input `[2,3,5]`, the resulting configuration should look like this (remember that user inputs are loaded from box 1 onwards, while box 0 is reserved for use by the machine itself):

```
boxes = [0,2,3,5]; counter = 0
```

```
instance Show BATConfig where
    ...
```

Problems 3–7. Now that we've defined BAT-computer configurations, we are ready to define how these configurations change when instructions are executed. Remember that user inputs are loaded into boxes 1 and upwards. The final output is found by counting the number of tokens in Box_1 if/when the program halts.

```
-- IMPLEMENTING THE BAT-computer
-- ============================
-- User inputs run from Box 1 onwards. Output is what ends up in Box 1.
-- Box 0 can be used by programs for calculations.
instance ProgrammableComputer BATConfig  where
    -- PROBLEM 3: initialise   :: Program -> [Input] -> cfg
    initialise ...
    -- PROBLEM 4: acceptState  :: Program -> cfg -> Bool
    acceptState ...
    -- PROBLEM 5: doNextMove   :: Program -> cfg -> cfg
    doNextMove ...
    -- PROBLEM 6: runFrom      :: Program -> cfg -> cfg
    runFrom ...
    -- PROBLEM 7: getOutput    :: cfg -> Output
    getOutput ...
```

The following function, `execute`, is included to help with testing. Running "`execute p xs`" should show the output generated when running program `p` with user input(s) `xs`.

```
execute :: Program -> [Input] -> Output
execute p ins = getOutput ((runProgram p ins) :: BATConfig)
```

Chaining programs together. Sometimes we want to run one program after another. The first program puts its output into Box_1, and then the second program operates on that value. We can do this simply by joining the two programs together – provided we remember to change the instruction numbers in the second program's `JEQ` instructions. This is what the `transpose` function should do.

Problem 8.

```
-- start a program at instruction n instead of 0.  In other
-- words, change Jump instructions from (J x y t) to (J x y (t+n))
-- and leave all other instructions unchanged.
transpose :: Int → Program → Program
transpose ...
```

Now that we know how to make sure the `JEQ` instructions refer to the right instructions when we join programs together, we write the "join" function itself. This is what the following (infix) function does. If you give it two programs `p1` and `p2`, then "`p1 *→* p2`" is the program equivalent to running `p1` and then `p2`.

Problem 9.

```
-- join two programs together, so as to run one
-- after the other
(*→*) :: Program → Program → Program
p1 *→* p2 = ...
```

Simple addition. Now that we've implemented the BAT-computer, it's time to write some programs. The first program, `adder`, should add the values in Box_1 and Box_2, and leave the result in Box_1.

Problem 10.

```
-- program to compute B1 = B1 + B2
adder :: Program
adder = ...
```

Copying from one box to another. Sometimes we need to move values from one box to another. The following function should implement this behaviour, i.e. it implements the assignment operation $\mathsf{Box}_n = \mathsf{Box}_m$.

Problem 11.

```
-- create a program to copy the contents of box m to box n (leave box m unchanged)
copyBox :: Int → Int → Program
copyBox m n = ...
```

And finally ... a more complicated addition function. Earlier you implemented the function $\mathsf{Box}_1 = \mathsf{Box}_1 + \mathsf{Box}_2$. This time, we want the user to be able to choose which boxes the input values are in. The program `addXY x y` should implement the function $\mathsf{Box}_1 = \mathsf{Box}_x + \mathsf{Box}_y$.

Problem 12.

```
-- program to compute B1 = Bx + By
addXY :: Int → Int → Program
addXY x y = ...
```

END OF ASSIGNMENT