CSCC24 2020 Winter – Assignment 4
Due: Friday, April 3, midnight
This assignment is worth 10% of the course grade.

In this assignment, you will implement in Haskell an interpreter for a toy language.

As usual, you should also aim for reasonably efficient algorithms and reasonably organized, comprehensible code.

Orlang

Orlang starts out as a "simple" (just you wait) imperative language. It has a fixed set of two integer variables, x and y. It has some common arithmetic operators and boolean operators. It has a while-loop. The detailed constructs are defined by the algebraic data types `Cmd`, `ExprNum`, and `ExprBool`; here is a brief orientation.

In human-friendly syntax, each command (statement) looks like one of:

- *var* := *exprnum*
- {*cmd*; ...} (sequential compound)
- **while** *exprbool* {*cmd*; ...}
- two more below

exprnum is an arithmetic expression and may use integer literals, the two variables, and some common arithmetic operators.

exprbool is a boolean expression and may use comparison between two *exprnum*'s, and some common boolean logical operators.

The wrinkle introduced by Orlang is nondeterminism! There are two more kinds of commands:

- *cmd* **or** *cmd* (nondeterministic choice)

  Example: $x := x + 1$ **or** $x := x + 2$

  An informal imagining of its behaviour is a multiverse story. The current universe is forked into two: in one universe, $x := x + 1$ is done; in another, $x := x + 2$ is done. (These two universes don't know of each other. We, the overseers, of course are aware that both exist.)

- **assert** *exprbool* (test and possibly fail)

  An informal imagining of its behaviour: The current universe survives unchanged (success) if the boolean expression evaluates to true; else the current universe disappears (failure).

Mind blowing happens when we use **or** inside a while loop that causes the loop to end earlier or later:

```
while x<3 and y==0 {
  x:=x+1;
  y:=1 or {};
}
```

Suppose $x = 0$ and $y = 0$ initially. In each iteration, two possibilities happen: y becomes 1 (so the loop exits), y stays as 0 (so the loop runs again unless $x \geq 3$. Then the program ends in 4 final universes:

- $x = 1, y = 1$
- $x = 2, y = 1$
- $x = 3, y = 1$
- $x = 3, y = 0$

Nondeterminism and conditional failure offer an interesting, high-level way to express search algorithms: Use **or** to initiate exploring two possibilities (add a loop for more possibilities like above); use **assert** later to check satisfaction. The final surviving universes are exactly the solutions. One of the test cases solves an equation by brute force this way. If Orlang supported more variables, more constructs, and more data structures, it could be used to express all search algorithms in this style. Some people have been thinking that this is the style of the next generation of programming languages, for the past 3+ generations.

The next section describes a formal model grounded in both math and code.

Monad for Orlang

Since both mutable variables and nondeterminism are involved, we expect to mix the state monad and the list monad. For the exact way to mix, the most important factor is that given one current state, nondeterministic choice leads to multiple parallel new states, and failure leads to none. So we define:

```
data OM a = MkOM (Variables -> [(Variables, a)])
```

in which `Variables` is my state, and it simply stores the two integers for x and y.

`OM` can be made an instance of Functor, Applicative, Alternative, and Monad. (In the Alternative instance, `empty` always fails, and `<|>` is for nondeterministic choice.) In addition, there can be two more primitives for reading and writing the two variables. There are all captured by this type class:

```
class (Monad m, Alternative m) => MonadOrlang m where
    -- Read x or y
    get :: Var -> m Integer
    -- Write x or y
    set :: Var -> Integer -> m ()
```

An interpreter for Orlang can be written entirely in terms of this class, i.e.,

```
interp :: MonadOrlang m => Cmd -> m ()
```

Then we can instantiate it to `OM` to run sample Orlang programs and get results!

Question 1 [7 marks]

Implement Monad and Alternative methods for `OM`. (The rest are already done: Functor and Applicative methods simply call Monad methods; but do take a look at how `get` and `set` are done.)

Question 2 [11 marks]

Implement the Orlang interpreter. Evaluators for number and boolean expression are already done for you (you already know how anyway).

Two functions `run0` and `run` are included to run your interpreter using `OM`.

(Real reason I introduce `MonadOrlang`: I will test your interpreter under my correct instance of `MonadOrlang`, so you won't lose marks here in case you have trouble with Question 1.)

End of questions.