CSCC24 2024 Summer – Assignment 4
Due: August 7, midnight
This assignment is worth 10% of the course grade.

Question 1: Parametricity (5 marks)

In parametricity.txt, complete the proof of: If $e :: \forall a.(\text{Integer} \to a) \to [a]$, then:
for all type A, for all $f :: \text{Integer} \to A$, $map\, f\, (e\, id) = e f$.
(id is the identity function: $\lambda x \to x$.)
It is OK if it is more convenient for you to end up with some other names for A and f.

Question 2: TryMe Interpreter (10 marks)

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

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

Code correctness (mostly auto-testing) is worth 90% of the marks; code quality is worth 10%.

TryMe is an imperative language that has mutable integer variables, integer-valued expressions, and exception throwing and catching. The detailed constructs are defined in TryMeDef.hs by the algebraic data types `Stmt`, `Expr`, and `Exception`. Here are the cases for statements in both familiar syntax and `Stmt` form:

familiar syntax	`Stmt` form
`var := expr`	`Assign var expr`
`{ stmt; ... }`	`Compound [stmt, ...]`
`try { stmts }`	`Try stmts [(exn, stmts), ...]`
`catch (exn) {stmts}`	
`catch ...`	

The cases for expressions are integer literal (can be negative), variable, addition, and division. The cases for exceptions are division by zero (`DivByZero`) and uninitialized variable (`VarUninit`). Some informal points on semantics:

- `Assign` initializes or modifies *var*'s value, whether or not it was initialized before.
- When looking up a variable's value, if it was not initialized before, this throws the `VarUninit` exception.
- When evaluating division, if the divisor is 0, this throws the `DivByZero` exception.
- When evaluating addition or division, evaluate the operands in the given order, e.g., evaluate the 1st operand first. This settles the question of which exception to throw if both operands would throw exceptions.
- `Try b0 [(e1, b1), ...]` runs `b0` and catches only the exceptions in the list, not other exceptions. Example: `Try b0 [(DivByZero, b1)]` catches `DivByZero` from `b0`, but not `VarUninit`.

If a handler such as `b1` throws an exception, it is not caught by this `try` block.

If the list mentions the same exception multiple times, only the first occurence matters, e.g., in `[(DivByZero b1), (DivByZero b2)]`, ignore `b2`.

Model

A concrete model of TryMe is a state transition function that also includes an `Either` result for exceptions and successes. The state can be a direct map from variables to values. (All variables are global, so we don't need a middle address layer.) Note that there is always a new state, even when an exception is raised—state changes are never lost upon exceptions.

```
data TC a = MkTC (Map String Integer -> (Map String Integer, Either Exception a))
```

The following type class `TryMeModel` summarizes the essential methods for any model and/or interpreter for TryMe:

```
class TryMeModel m where
    -- Give an answer.
    pure :: a -> m a
    -- Sequential composition.
    (>>=) :: m a -> (a -> m b) -> m b
    -- Throw the given exception.
    raise :: Exception -> m a
    -- Try, then report exception or success.
    reifyException :: m a -> m (Either Exception a)
    -- Init/Write a variable.
    putVar :: String -> Integer -> m ()
    -- Read a variable. Throws VarUninit if not found.
    getVar :: String -> m Integer
```

Implement those methods for `TC` in TryMe.hs

Interpreter

Implement the TryMe interpreter in TryMe.hs:

```
interp :: TryMeModel m => Stmt -> m ()
```

The polymorphism forces you to code to the `TryMeModel` interface, but the benefits are better focus and being higher-level. (Another is independent testing when marking.)

Two functions `run` and `runWith` are included to run your interpreter using `TC`: `run` starts with no variable initialized, `runWith` starts with a given state.

End of questions.