COS 441 Assignment #5: Monads

Most of the materials for this assignment may be found in a5-materials.tar.gz. Download that bundle from the course web site. Email the bundle back to the TA by the assignment deadline.

Part I

Examine the file monad.lhs. Answer the questions about monads and monad laws within the file.

Part II

Examine the file Expressions.hs. This file depends upon the file State.hs (which is extremely similar to but not identical to the file State.hs used in the previous assignment). Expressions.hs contains an expression language with several features:

- String expressions as well as Integer expressions (a value is either a string or an integer)
- Division expressions (including possibly division by zero, which raises an exception)
- PrintThen e1 e2, which prints the string that e1 evaluates to and then returns e2

Your job is to implement an evaluator function "eval" for the Expression language with this type:

```
type Result = (String, Either Value Exception)

eval :: State -> Exp -> Result
```

Either is a type in the standard Haskell library. It's definition is:

```
Either a b = Left a | Right b
```

So, if evaluation of your expression raises an exception e, you should return "Right e" as the result of evaluation. If evaluation of your expression does not raise an exception, you should return "Left v" where v is the value the expression computes. In either case, you should also return the string s that is printed during the course of evaluation.

In order to implement the evaluator, you must make use of monads and do notation to help manage the "plumbing" that arises. There is a spot in Expressions.hs for you to define your own monad to help you write the evaluator code. The first thing to do to define your monad is to choose the type of the monad. Please call your monad type "Effects a."

```
newtype Effects a = ....
```

Then proceed to define the return and bind (ie: ">>=") functions for the monad. You should check that your monad obeys the monad laws 1-3 discussed in part I of this assignment, but you do not have to hand in a formal proof that they do.

As we saw in lecture, it is often useful to define some auxiliary functions (in addition to return and bind) that help a programmer program with a monad. For example, in slides17, slide number 13, from the course website, we defined the function "printme" with type String -> Out a. This function was used to print a string within a monad. You may find it useful to create another such function. Other examples of such auxiliary functions are "getState" and "setState" used with the storage monad. Since this evaluator uses exceptions, you will want to think carefully about what kinds of auxiliary functions you need to help you create and raise exceptions within the monad. For example, we recommend you create a function "raise" that will throw an exception in your monad. Moral of the story: Think carefully about how to structure your code so it is clean and clear. If there are repeated idioms that can be factored out into clean functions, you should do so. Your evaluator will not only be judged on its correctness but also on the clarity of its code.

Keep the following constraints in mind when implementing and testing your evaluator:

- If division by zero occurs, the divide_by_zero_exception should be thrown
- If an operator is applied to the wrong type of value the bad_type_exception should be thrown. Eg: if an expression tries to add strings or concatenate integers, throw the bad_type_exception
- If an expression refers to a variable that does not have a value in the current state, the undefined_variable_exception should be thrown
- Arguments of expressions are evaluated left-to-right. So, assuming that a and b are integer constants, the expression:
  (PrintThen "X" a) :+: (PrintThen "Y" b)
  should produce the string "XY" and return the integer value resulting from a+b.
- If evaluation encounters an exception, no further printing should occur but the string printed so far should be retained in the final result. For example:
  (PrintThen "X" a) :/: (num 0)
  should produce the string "X" as a final result (not "") as well as raising a divide_by_zero_exception

Send email to the cos441 mailing list for clarification on the semantics of evaluation of expressions.