CSC324 Assignment 2: An Interpreter in Haskell

In this assignment, we will write an interpreter for a functional language embedded in Haskell. This interpreter is for a language we call Orange, which is very close to MandarinBasic from the first assignment, and the interpreter is similar to the one we discussed in class.

Because Haskell is a strongly typed language, implementing an interpreter in Haskell will help you better-understand the *types* of objects that are being acted upon in an interpreter. This assignment is also an opportunity to work with pattern matching, error propagation, and Haskell in general.

Before we start, remember these general guidelines for all labs and assignments:

- You may not import any libraries or modules unless explicitly told to do so.
- You may write helper functions freely; in fact, you are encouraged to do so to keep your code easy to understand.

Breaking any of these above rules can result in a grade of 0.

- Code that cannot be imported (e.g., due to a syntax error, compilation error, or runtime error during import) will receive a grade of zero! Please make sure to run all of your code before your final submission, and test it on the Teaching Lab environment (which is the environment we use for testing).
- The `module (...) where` code in your files is very important. Please follow the instructions, and don't modify those lines of code! If you do so, your code will not be able to run and you will receive a grade of zero.
- Do not change the default Haskell language settings (i.e. by writing an additional line of code in the first line of your file)

Starter code

- `A2.hs`
- `A2Types.hs`
- `A2StarterTests.hs`
- `haskellify.rkt`

You will only be submitting `A2.hs` and not any of the other files. Do not make any modifications to `A2Types.hs`. We'll be supplying our own version to test your code.

The language `Orange`

The language `Orange` is an expression-based language that is eagerly evaluated left-to-right. The language supports three types of values: booleans (T/F), numbers, and pairs. A `Orange` expression grammar is described using the `Expr` abstract data type in the `A2Types.hs` file.

```haskell
data Expr = Literal Value           -- literal values
          | Plus Expr Expr          -- builtin "plus" function
          | Times Expr Expr         -- builtin "times" function
          | Equal Expr Expr         -- builtin checks for equality
          | Cons Expr Expr          -- builtin "cons" function that creates a pair
          | First Expr              -- builtin "first" function that obtains the first element of a pair
          | Rest Expr               -- builtin "rest" function that obtains the second element of a pair
          | Var String              -- variable names
          | If Expr Expr Expr       -- if expressions
          | Lambda [String] Expr    -- function definitions
          | App Expr [Expr]         -- function applications

data Value = T | F                  -- booleans true an dfalse
           | Num Int                -- integers
           | Empty                   -- the empty list
           | Pair Value Value       -- pairs
           | Closure [String] Env Expr -- closures
           | Error String           -- errors
```

Your task for this assignment is to write an interpreter for this language. Recall that an interpreter will take an environment and an expression, and determine the value of the expression.

```
eval :: Env -> Expr -> Value
```

Notice that in addition to the three types of values described earlier, our definition of `Value` also describes two additional value constructors `Closure` and `Error`. Here, `Closure` describes a closure (resulting from a function expression), and `Error` describes an error when evaluating an expression.

In the rest of this handout, we will describe the behaviour of each kind of expression. Read this file carefully. We recommend writing test cases as you go along (e.g. in the `A2StarterTests.hs` file). Doing so will help you engage with the handout and stay focused, while also saving you some time down the road!

Once again, remember to *trust in the recursion*. Tackle each kind of expression one at a time. The order that we provided is a fairly good way for you to get started.

Warmup task before you start (optional, but highly recommended!)

`Orange` is a programming language. It might not look like the programming languages you've seen so far, though, because you can only write `Orange` programs by manually constructing a Haskell data-structure that represents the Abstract Syntax Tree (AST). In most programming languages, there is a *parser* which reads source code and turns it into an AST for you.

In this warmup task, you will bridge this gap by writing (1) a Haskell function that turns an `Orange` expression into equivalent Racket code, and (2) a Racket function that turns the Racket equivalent of an `Orange` expression into the corresponding Haskell data structure.

While these functions will not be tested, we *highly* encourage you to complete them as they will be invaluable during testing. Function (1) will allow you to double check your interpreter against Racket's evaluation, while (2) will allow you to write sophisticated test cases in a much easier to use syntax. Both functions together should allow you to answer most questions you have about test cases. **We will not answer questions about test cases that can be answered via this warm-up task.**

Complete `racketifyExpr` in `A2.hs`. Much of the code has been completed for you. Note that `First` and `Rest` should map to `car` and `cdr` in Racket.

Complete `haskellify` in `haskellify.rkt`. Again, much of the code has been completed for you.

Once you have completed these functions, you can approach test cases as follows. To check if a test case for your `eval` function is correct, you can run it through `racketifyExpr`, run the output Racket code, and check whether your code has the same output. For example,

```
racketifyExpr (Plus (Literal $ Num 3) (Literal $ Num 10))
-- outputs "(+ 3 10)"; run this in Racket and compare outputs!
```

To write more sophisticated test cases, you can write equivalent Racket code and run it through `haskellify` to generate the corresponding Haskell data structure. For example,

```
(haskellify '(+ 3 10))
; outputs "(Plus (Literal (Num 3)) (Literal (Num 10)))"; use this as a test case in Haskell
```

Note. While this works very well for test cases where evaluation succeeds normally, it won't help with checking how your evaluator works with errors because Orange and Racket handle errors differently (i.e, Racket will throw an exception, while in Orange you are expected to return a specific Error value). You can still use `haskellify` to help write test cases for these, but you will have to verify your results manually.

It is strongly recommeneded to complete this warmup task. It should not take very long and will make your life with writing tests easier. Ask questions if you need help with this task! Once again, we will not answer questions like "is this test case correct" or "does this output make sense for this test case" if they can be answered by completing this task.

How to evaluate `Orange`

Literal values

Literal values in this expression evaluate to the following:

```
*A2> run (Literal T)
T
```

Like we discussed at the end of lecture 4, we use **pattern matching** to destruct a Haskell value of type `Expr`, so that we can get to the `v` inside the `Expr` value `(Literal v)`.

We will only allow literal booleans, numbers and pairs. Attempting to create a literal closure or a literal error should produce an `Error` with an appropriate message described towards the end of the handout.

Plus, Times

Expressions involving `Plus`/`Times` are expected to take two subexpressions that evaluates to numbers, and produce a number corresponding to the sum/product of the two numbers.

If either subexpression does *not* evaluate to a number, then return `Error` with an appropriate message described towards the end of the handout.

```
*A2> run (Plus (Literal $ Num 3) (Literal $ Num 4))
Num 7
```

(Recall that in Haskell, the expression `a $ b c` is equivalent to `a (b c)`. You can think of `$` as an infix operator for function application. If that doesn't make sense, think of `$` as a fancy way of avoiding brackets.)

We wrote some code for you to handle the `Plus` case, so that you can see an example of `case ... of ...` expression in Haskell. In particular, we can pattern match the **values** returned from `(eval env a)`. In the example code we are pattern matching the value of the tuple `((eval env a), (eval env b))`. Please see a few examples here http://learnyouahaskell.com/syntax-in-functions, particularly the section on "Case expressions".

You will need to write your own new patterns and bodies for the `eval` function from here onwards.

Equal

This expression takes two subexpressions. In most cases, use the Haskell `==` to check if the values of the subexpressions are equal. If so, return the boolean value `T`. Otherwise, return `F`. (Note that if you try to return Haskell "true" and "false" values, your interpreter will not type check!)

The only exception is if either of the subexpressions return an error. If so, return the first error encountered.

```
*A2> run (Equal (Literal $ Num 3) (Literal $ Num 4))
F
*A2> run (Equal  (Plus (Literal $ Num 3) (Literal $ Num 4)) (Literal $ Num 7))
T
```

Cons, First, Rest

A `Cons` expression takes two subexpressions, and creates a `Pair` out of the resulting expressions. Again, the exception is if either of the two sub-expressions result in an `Error`. If so, return the first error encountered.

The `First` and `Rest` expressions take one subexpression. If a pair is returned, extract either the first/second element of the pair. Otherwise, return an error.

```
*A2> run (Cons (Literal $ T) (Literal $ F))
Pair T F
*A2> run (First (Cons (Literal $ T) (Literal $ F)))
T
*A2> run (Rest (Cons (Literal $ T) (Literal $ F)))
F
```

If expressions

Like in Racket, an `If` expression takes three subexpressions `(If cond expr alt)`. First, the *condition* subexpression `cond` is evaluated. If an error occurs during the evaluation of `cond`, propagate that error. Otherwise, we will use the rule that unless condition subexpression evaluates *exactly* to `F`, then we will evaluate `expr`. If the condition subexpression evaluates to `F`, evaluate `alt`. (This is consistent with how Racket's `if` expressions work.)

Note that in this expression, *only one branch of the "If" expression should be evaluated!*. Be careful about the order of evaluation. It would be incorrect to evaluate both branches of the `If` expression, and then determine which value to return.

```
*A2> run (If (Literal T) (Literal $ Num 3) (Plus (Literal T) (Literal F)))
Num 3
```

Variable lookup

A `Var` identifier should evaluate to its corresponding value in the environment, similar to what we did in the previous lab in Racket. Here, you may find the function `Data.Map.lookup` helpful.

One thing to note is that `(Data.Map.lookup name env)` does not return a `Value`. Instead, it returns `Maybe Value`, to account for the case where the identifier name is not defined in the environment. We will discuss the `Maybe` type constructor in week 5 and week 6, so the starter code provides the pattern matching code that you need to decompose the lookup result. The comments in the starter code in this section should be helpful.

If the variable name does not appear in the environment, return an error.

Function expressions

A function expression takes a list of identifiers, a body expression, and evaluates to a closure like discussed in lecture.

Unlike the Lambda Calculus++ language from lecture 4, Orange allows functions with zero or more arguments. As examples, these are all valid expressions in Orange, represented as a value of type `Expr` in Haskell:

```
-- a function that takes 0 parameters and returns `T
(Lambda [] (Literal T))

-- a function that takes 2 parameters and returns the second one
(Lambda ["x", "y"] (Var "y"))

-- a function that takes 2 parameters adds them
(Lambda ["x", "y"]                 -- these are the parameter names
        (Plus (Var "x") (Var "y")) -- this is the function body
)
```

Here is what should happen when we evaluate an identity function in Orange:

```
*A2> run (Lambda ["x"] (Var "x")) -- an identity function
Closure ["x"] (fromList []) (Var "x")
```

Like discussed in lecture 4, evaluating a `Lambda` expression should yield a closure containing the same information as in the function expression, plus the environment at the time to function was created. Our `Closure` constructor contains

- The list of parameters (e.g. `["x"]` from above)
- The environment (e.g. `(fromList [])` from above, which Haskell's representation of a Data.Map value)
- The body expression (e.g. `(Var "x")` from above)

Note that the body expression should not be evaluated until it is called! In other words, the following expression should *not* evaluate to an `Error`, even though evaluating `(Var "y")` will fail:

```
*A2> run (Lambda ["x"] (Var "y"))
Closure ["x"] (fromList []) (Var "y")
```

The list of parameters in the function expression should be unique: i.e., two parameters in the same `Lambda` should *not* have the same name. Otherwise, an error should be produced.

Function application

Like discussed in class, evaluating the function application `(App fnExpr argExprs)` involves several steps:

1. Evaluate `fnExpr`. This should result in a `(Closure params cenv body)`, otherwise there is an error. At this point, you can also check that the number of parameters is the same as the number of argument expressions, and return an `Error` otherwise.
2. Evaluate each of the `argExprs`. If any of these argument expressions evaluates to an error, return that error instead of continuing execution.
3. Finally, evaluate the `body` expression from the closure from step 1.

Pay close attention to the environment that you use (and/or construct) at this point. You may find `Data.Map.insert` helpful in determining what to do with the parameter names and arguments.

```
*A2> run (App (Lambda ["x"] (Var "x")) [Literal T])
T
```

Errors

The following errors should be reported:

- `Error "Literal"`: occurs when attempting to create a literal closure or literal error.
- `Error "Plus"`: occurs when an expression `(Plus a b)` has either `a` or `b` not evaluating to a number.
- `Error "Times"`: occurs when an expression `(Times a b)` has either `a` or `b` not evaluating to a number.
- `Error "First"`: occurs when an expression `(First e)` has `e` not evaluating to a pair.
- `Error "Rest"`: occurs when an expression `(Rest e)` has `e` not evaluating to a pair.
- `Error "Var"`: occurs when an identifier `(Var name)` does not exist in the environment.
- `Error "Lambda"`: occurs when a function expression contains two or more parameters with the same name.
- `Error "App"`: occurs when an application `(Apply fnExpr argExprs)` has `fnExpr` not evaluating to a closure, *or* if the list of expressions `argExprs` is of different length than the list of parameter names.

Moreover, we expect that the *first* error encounters in a left-to-right, eager evaluation order will be the one returned:

```
*A2> run (Plus (Literal $ Num 3) (Times (Literal $ Num 3) (Literal T)))
Error "Times"
*A2> run (App (Lambda ["x"] (Var "y")) [Literal T])
Error "Var"
*A2> run (App (Lambda ["x"] (Var "y")) [Literal T, Literal T])
Error "App"
*A2> run (If (Literal T) (Literal $ Num 3) (Plus (Literal T) (Literal F)))
Num 3  -- no error!
```

Getting Started

It is normal in this course to spend a long time thinking before writing code. If you begin writing code right away, you may end up writing a lot of code that serves little to no purpose. It's okay to work with examples for a long time. It's okay to think for a long time about a very short piece of code. It's also okay to throw out large amounts of prototype code and as you understand the problem better.

As in previous assignments and labs, code incrementally and test incrementally, so that you always have working code.

If you are working with a partner, we highly recommend pair coding. Pair coding is especially helpful for this course, since it is easy to make both syntax and logical errors while coding. Handle one expression type at a time, and expand your tests as you go along.