CSE 341, Winter 2014, Assignment 4
Octopus Interpreter
Due: Friday Feb 7, 10:00pm

Assignment updated Feb 1

The purpose of this assignment is to give you experience with writing a larger program in Haskell, and also with writing interpreters. All your code should be in the functional part of Haskell (no monads), except for the unit tests and for the final read-eval-print loop (Question 8).

Points: 60 points, plus up to 6 points extra credit.

Start early! This assignment doesn't involve writing that much code, but you'll need to understand and extend existing Haskell code, and to understand thoroughly the semantics of closures in Racket. Plus debugging an interpreter will be a new skill.

You can use up to 4 late days for this assignment.

Turnin: Turn in one file: `OctopusInterpreter.hs`, which should include all your functions and unit tests. If you do the String extra credit problem (Question 10), also turn in your parser file. If you do the dynamic scoping extra credit problem (Question 9), turn in another version of the interpreter called `OctopusInterpreterDynamicScope.hs`.

You don't need to turn in sample output — the unit tests are enough for those. As usual, your program should be tastefully commented. Style counts! In particular, think about where you can use pattern matching and higher order functions to good effect to simplify your program; and avoid unnecessary repeated computations.

Overview: The OCTOPUS programming language is a small subset of Racket, but even though it leaves out many of Racket's features, it is still among the most expressive of the invertebrate programming languages.

Every OCTOPUS program is also a legal Racket program. The data types in OCTOPUS are integers, booleans, symbols, lists, and functions. There are no side effects. Functions are defined using `lambda`, which has exactly the same meaning as in Racket — it creates a lexical closure. The other special forms are `let`, `quote`, `if`, and (if you do the recursion extra credit question) `letrec`. One important restriction is that there is no `define` special form — to create and bind new variables, use `let` or `lambda`. Some other minor restrictions are that `let` and `lambda` always have just one expression in the body (since there are no side effects, having multiple forms wouldn't be useful). The functions `+`, `-`, and `*` take exactly two arguments. Finally, lists are always proper lists — no dotted pairs like `(2 . 3)`.

There are two starter files, linked from the class website: `OctoParser.y` and `OctopusInterpreter-starter.hs`. `OctoParser.y` is a parser for OCTOPUS, written using the Happy parser generator (`http://www.haskell.org/happy/`). Unless you do the string extra credit question, you shouldn't need to modify it at all. Just run the Happy parser generator from the command line:

```
happy OctoParser.y
```

This should generate a file `OctoParser.hs` that is the parser. (This `.hs` file isn't intended to be particularly human-readable.)

Download `OctopusInterpreter-starter.hs` and rename it to `OctopusInterpreter.hs`. Load it into Haskell and run the first few unit tests using `run`, to make sure things are working OK. The interpreter will automatically load the parser (make sure they are in the same directory).

The key things you need from the parser are the types `Environment` and `OctoExpr`, and a function `parse` whose type is `String -> OctoExpr`. You'll need to know the definition of these types in writing your interpreter — here they are (or look in `OctoParser.y`):

```
-- An environment is a list of (name,value) pairs.
type Environment = [(String,OctoExpr)]

{- Declarations of the datatype for Octopus data.  The constructors
used in data produced by the parser are OctoInt (Octopus integers),
OctoSymbol (Octopus symbols, or atoms), and OctoList (lists).  The
remaining two types, OctoClosure and OctoPrimitive are not actually
used by the parser, just the interpreter.-}

data OctoExpr
      = OctoInt Int
      | OctoSymbol String
      | OctoList [OctoExpr]
      | OctoClosure [String] Environment OctoExpr
      | OctoPrimitive String
      deriving (Show, Eq)
```

Experiment a bit with this. For example, `parse "(+ 3 4)"` should return `OctoList [OctoSymbol "+",OctoInt 3,OctoInt 4]`.

Then begin adding functionality to the parser, as described below. Most of the calls to the unit tests are commented out — enable more and more of them as you add functionality. You will also need to add unit tests for the primitive functions — right now there is only a test for `+`. The other tests are enough to test the other functionality, although you are welcome to add more if you want.

You don't need to do error checking in your interpreter. (The starter program does include a little error checking, for example for parse errors and unbound variables, which helps with debugging.)

1. (12 points) Add new primitives for `-`, `*`, `cons`, `car`, `cdr`, and `equal?`. Add unit tests for these. To add these primitives, write new Haskell functions `octominus`, `octotimes`, and so forth, following `octoplus` as a model, and add them to the list of primitives (defined just before `octoplus` in the starter code). You shouldn't need to modify the `eval` function at all for this question.

2. (10 points) Write a function `octoshow` that turns any OCTOPUS expression (represented as data of type `OctoExpr`) into a string. Here are a few examples:

   ```
   OctoInt 7    =>  "7"
   OctoList [OctoInt 1, OctoInt 2, OctoInt 3]    =>  "(1 2 3)"
   OctoList [OctoSymbol "squid", OctoSymbol "clam"]    =>  "(squid clam)"
   OctoList [OctoSymbol "quote", OctoSymbol "squid"]    =>  "'squid"
   ```

   Modulo white space, `parse` and `octoshow` are inverses. Note that for lists there shouldn't be an extra space before the right parenthesis. (Hint: the Haskell function `unwords` may be useful.) For example:

   ```
   octoshow $ parse "7"    => "7"
   octoshow $ parse "(1 2  3   )"    =>  "(1 2 3)"
   octoshow $ parse "(+ 1 (* 2  3))"    =>  "(+ 1 (* 2  3))"
   octoshow $ parse "'(1 2 3)"    =>  "'(1 2 3)"
   octoshow $ parse "'squid"    =>  "'squid"
   ```

   You won't encounter an `OctoClosure` or an `OctoPrimitive` in parser output — these are just used internally in the interpreter. So you can show them just as `"<closure>"` and `"<primitive>"` respectively. (You can return something more elaborate if you wish, but it's not required. For example the sample solution shows the primitive for `+` as `"<primitive +>"`.)

3. (8 points) The starter interpreter includes code to handle applying primitive functions but not user defined functions (i.e. ones written using `lambda`). Fill this in (search for the text `TO BE WRITTEN`). The tests `test_lambda1`, `test_lambda2`, and `test_shadow` should now succeed. The code for this is just a couple of lines in the sample solution, but you might find it a bit tricky to figure out. Be sure and read the comment before the skeleton of `apply` regarding what needs to be evaluated where. You just need to replace the `error ...` part of the definition of `apply` for this question; you shouldn't need to modify the final case of `eval` (which is where `apply` is called from).

   After you have `lambda` working, add the `null?` function to the global environment — it's already written, but you need to change the definition of `global_env`. Search for `global_env = primitive_env` in the starter code, and replace that with the commented-out code that follows. (This was commented out initially, since it requires lambda to work.)

4. (5 points) Add code to handle the `if` special form. Implement this directly — this should be straightforward. This will involve adding a new case to the `eval` function. The tests `test_if_true` and `test_if_false` should now succeed.

5. (5 points) Add a function `not` to the global environment. This should be defined in Octopus (like `null?`) rather than written as a primitive. (Search for `null?` and follow that as an example — do not modify the `eval` function by adding a special case for `not`.)

6. (10 points) Add code to handle the `let` special form. There are at least two ways to do this. One is by implementing it directly, as you did with `if` in Question 4. Another is to define it as a derived expression in terms of `lambda` — that is, the case of your `eval` function that handles `let` should produce a new expression using `lambda`, and then evaluate that. For example, suppose you are evaluating this `let` expression:

   ```
   (let ((x 5)
         (y 10))
     (+ x y))
   ```

   Produce the following expression that uses `lambda`, and evaluate that. Notice that the `lambda` takes care of all of the work of evaluating the bindings for `x` and `y` in the proper environment, making a new environment, and evaluating the body of the `let` .

   ```
   ( (lambda (x y) (+ x y)) 5 10)
   ```

   Using either technique is fine, but please be sure you understand how they both work. In either case you will need to add a new case to the `eval` function. (You can't just make it a new primitive since it's a special form, and you need to control how the parts are evaluated.)

7. (5 points) Add a primitive to implement an Octopus `eval` function. This should work like the one-argument version of `eval` in Racket, in other words, the one without the namespace argument. Again as with Racket, the expression should be evaluated in the global namespace in that case (for the Octopus interpreter, this is stored in `global_env`). Hint: first evaluate the argument in the current environment. (Defining `eval` as a primitive will automatically take care of doing this.) Then evaluate the result again in the global environment. There are some unit tests that check this.

8. (10 points) Finally, add a simple read-eval-print loop, using Haskell's IO functions (monads). The loop should get a line from the keyboard, parse it, evaluate it, convert it to a string using `octoshow`, and print it out. Keep looping until the user types a blank line.

   There is a compiled version of the Octopus interpreter on attu, if you want to try the read-eval-print loop: invoke it from the shell using `~borning/octopus`.

9. **Extra Credit.** (1 point) Racket (and OCTOPUS) use static scoping. Older Lisps, and some other older languages, use *dynamic scoping*. To look up a name, look in the current function, then look up the calling stack until you find it (or fall off the end). Even though this is a major change in the semantics of the language, it's easy to convert your OCTOPUS interpreter to use dynamic scoping. Do that (turn in a separate file named `OctopusInterpreterDynamicScope.hs`). Include a test case that shows that it is working. In addition, some of the existing tests will fail with dynamic scoping. In a comment at the top of your program, indicate what your new dynamic scope test is, and which of the existing tests fail and why.

   As an example, this code will give an error with Racket and OCTOPUS, but works with dynamic scoping:

   ```
   (let ((f (lambda (x) (+ x y))))
     (let ((y 10)) (f 20)))
   ```

   In OCTOPUS, this gives an error — `y` is unbound in the body of `f`. But with dynamic scoping, it finds the binding to 10. (There are problems and subtle bugs that arise with dynamic scoping — you can end up with variables captured that you didn't intend — and it's harder for both humans and compilers to reason about. But you should know the concept.)

10. **Extra Credit.** (2 points) Add a string datatype to OCTOPUS, and add a primitive `string-append` function. After this is done, you should be able to evaluate expressions like this:

    ```
    (string-append "giant" "squid")
    ```

    To simplify this, you can restrict `string-append` to exactly two arguments, and also not handle strings with embedded double quotes (so no `"oyster\"clam"`). Provide an appropriate unit test. For this extra credit problem, you'll need to modify the parser as well as the interpeter. (You should at least skim Chapter 2 of the Happy parser documentation.)

11. **Extra Credit.** (3 points) The `let` special form allows you to define functions by binding a lambda to a variable — but since the function name isn't known in the lambda, the function can't be recursive. Racket has a variant of `let`, called `letrec`, to support just such recursion. Add a special form for a limited version of `letrec`, which allows for binding only a single variable to a function of one argument. This is enough to open up lots of additional possibilities for functions — the unit tests include factorial and count functions. And if you want a function of two arguments, just write a curried version — there is another unit test with a curried version of `map`.

    Racket handles `letrec` by creating the variables in the `letrec`, temporarily binding them to `#<undefined>`, and then going back and splicing in pointers to their values. However, we're not using side effects. One approach (which would also work if we were writing the OCTOPUS interpreter in the functional subset of Racket) is to use the Y combinator, which allows you to unroll a recursive function into a non-recursive version. We'll talk about the Y combinator in lecture. There is also an example Racket program on the assignments web page that gives a definition of the Y combinator, a `letrec` definition of a recursive function, and a translation of the recursive function into a non-recursive version . . . you can use this as a model for your solution. Another approach — which is considerably simpler, and also handles functions with an arbitrary number of arguments — relies on the fact that Haskell uses lazy evaluation (so it wouldn't work in Racket). Either approach is acceptable for this problem. If you take the second approach, explain in a comment why it wouldn't work in Racket.