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 10).

Points: 75 points, plus up to 8 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 12), also turn in your parser file. If you do the dynamic scoping extra credit problem (Question 11), 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 expression is also a legal Racket expression, and if the expression evaluates without error in Racket, it should also evaluate without error in Octopus and the results should be the same. 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 `quote`, `if`, `cond`, `let`, and `letrec`. One important restriction is that there is no `define` special form. Instead, to create and bind new variables, use `let`, `letrec`, or `lambda`. Another restriction is that `let`, `letrec`, and `lambda` always have just one expression in the body (since there are no side effects, having multiple forms wouldn’t be useful). 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/](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 `OctoValue`, and a function `parse` whose type is `String -> OctoValue`. You’ll need to know the definition of these types in writing your interpreter — look in `OctoParser.y`.

1
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 null?, not, and 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, unless you add it for the extra credit question. (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 modify the eval function for this question.

2. (10 points) Write a function octoshow that turns any Octopus value (represented as data of type OctoValue) into a string. Here are a few examples:

   OctoInt 7 => "7"
   OctoBool False => "#f"
   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="#f" => "#f"
   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. You can show them just as "<closure>" and "<primitive +>" (or whatever the name of the primitive is) respectively. (You can return something more elaborate for closures if you wish, but it’s not required.)

   There are unit tests for octoshow that you should uncomment before starting on this part. (They don’t test having extra spaces in the input, or OctoClosure — you can add some tests for those if you want but it’s not required.)

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 involving lambda should now succeed. The code for this is just one line 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). Hint: the zip function may be handy.

   After you have lambda working, the null? function should work. (It’s already defined in the global environment.) Add unit tests for it however.
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 four if tests 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.) Add unit tests for not.

6. (5 points) Add code to handle the let special form. Recommendation: implement it directly, as you did with if in Question 4.

An arguably more interesting approach is to define it as a derived expression in terms of lambda — that is, the case of your eval function that handles let would produce a new expression using lambda, and then evaluate that. However, this won’t be as useful when you get to letrec. But you should still understand how the derived expression technique works. For example, suppose you are evaluating this let expression:

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

You’d 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)

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) Implement cond. You can assume that the last expression in the cond will always be (else expr). For full credit, you should implement this by transforming the cond into a set of nested if expressions, which you then evaluate. Here are some examples of translated expressions:

- (cond (else (+ 3 4))) → (+ 3 4)
- (cond ((equal? 2 3) 8) (else (+ 3 4))) → (if (equal? 2 3) 8 (+ 3 4))
- (cond ((equal? 2 3) 8) ((equal? 10 10) 100) (else (+ 3 4))) → (if (equal? 2 3) 8 (if (equal? 10 10) 100 (+ 3 4)))

Notice how this is much like implementing this as a macro in Racket. Hint: write a helper function cond_to_if that takes a list of cases in a cond and returns the equivalent expression.

9. (5 points) Implement letrec. You should now be able to run all the tests using recursive functions.

Hints: this should be trivial to implement by copying and modifying your code for let. In the sample solution, both let and letrec use the same helper function eval_let_bindings that computes the bindings for the variables in a let or letrec, so that there isn’t any duplicated code.

If you’re confused, go back and review how let works. For let, we start with the enclosing environment of the let — let’s call this enclosing_env. Also, let’s call the variables that are bound by the let expression let_vars. We extend enclosing_env with bindings for each of the let_vars to yield
The expressions that define the let_vars are evaluated in enclosing_env. Finally, we evaluate the expression in the body of the let in extended_env.

letrec can be implemented in exactly the same way, except that the expressions that define the letrec_vars are evaluated in extended_env rather than enclosing_env. This may seem circular, since we are using extended_env in defining extended_env — and it is, but it works because Haskell is lazy.

10. (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. This compiled version includes code for the string extra credit problem (Question 12).

11. 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:

\[
\begin{align*}
\text{(let ((f (lambda (x) (+ x y))))} \\
\text{\quad (let ((y 10)) (f 20)))}
\end{align*}
\]

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.)

12. 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:

\[
\begin{align*}
\text{(string-append)} \\
\text{\quad (string-append "A" "Giant" "Squid")}
\end{align*}
\]

To simplify this, you don’t need to handle strings with embedded double quotes (so no "oyster\\"clam"). There are commented-out unit tests for this question in the starter program already. For this extra credit problem, you’ll need to modify the parser as well as the interpreter. (You should read the appropriate parts of Chapter 2 of the Happy parser documentation.)

13. Extra Credit. (1 point) Are there examples of using letrec that evaluate correctly in Octopus but not in Racket? If so, give an example; if not, argue why not.

14. Extra Credit. (up to 4 points) Add an additional feature (or features) to your Racket interpreter. Here are some suggestions, but you can also do something of your own choosing.

- It’s a nuisance to need to define functions just using letrec. Add support for define at the top level of the read-eval-print loop.
• Once you add `define`, add support for `(load "filename")`, also just at the top level of the read-eval-print loop. Naturally, a file should be able to contain further `load` commands as well as function definitions.

• Add some exception handling, so that rather than just crashing, if you have an error in an expression in the read-eval-print loop the system prints out a helpful message and continues on with the next prompt.