About Lab 6

So where are we with the interpreter project?

I changed the due date of Lab 5B to Friday April 6. If you haven't completed it yet you should get it done; what is coming next is much more interesting.

Just to remind you, in Lab 5B you created a module env.rkt. This module has data types for our interpreter's environment. This includes constructor

(extended-env syms vals old-env)

Lab 5B also calls for a lookup function

(lookup env sym)

that returns the value bound to symbol *sym* in environment *env*.

In the last in-person class we had I outlined the first two parts of Lab 6. In Labs 6 and 7 we implement step by step a language I call MiniScheme which has most of the functionality of real Scheme. Our interpreter design calls for two modules. One is module parse.rkt that contains a function (parse input). This function takes an expression (such as '(let ([f (lambda (x) (+ x 1))]) (f 5))) and builds a parse tree that represents the expression. Module parse.rkt also contains definitions of all of the parse tree datatypes. The other module is interp.rkt. This contains function (eval-exp tree env) that evaluates a parse tree in a particular environment.

Here is code I wrote on the board for the first two steps: MiniSchemeA (which contains only numbers) and MiniSchmeB (which also contains symbols).

The parser is

```
(define parse (lambda (input)
        (cond
            [(number? input) (new-lit-exp input)]
            [(symbol? input) (new-var-ref input)]
            [else (error 'parse "Invalid syntax ~s" input)])))
```

The interpreter for MiniSchemeB has this for eval-exp:

```
(define eval-exp (lambda (tree env)
        (cond
                [(lit-exp? tree) (lit-exp-num tree)]
                [(var-ref? tree) (lookup env (var-ref-sym tree))]
                [else (error 'eval-exp "Invalid tree: ~s" tree)])))
```

The parser file should also contain definitions of the tree data types:
lit-exp: constructor new-lit-exp, recognizer lit-exp? and getter lit-exp-num
var-ref: constructor new-var-ref, recognizer var-ref? and getter var-ref-sym

The parser should provide those symbols and the interpreter should require the parser.

These are the simple cases; be sure you understand how they fit together.

Suppose we make a new file called MiniScheme.rkt and have this require env.rkt, parse.rkt and interp.rkt. In this file we might say

(define T (parse 23))

This makes T into a parse tree: ('lit-exp 23)
Lab5B asks you to build a top-level environment called init-env. If we run (eval-exp T init-env) it uses the lit-exp getter to pull the number out of T and returns it: 23.

Lab5B asks you to provide a few variable bindings in init-env. Here is where we can use them. Let's assume init-env binds symbol x to 10. If we define T1 in the MiniScheme.rkt file as (parse 'x), then (eval-exp T1 init-env) first pulls the symbol x out of the tree T1, then looks it up in the envirionment init-env to get value 10.

MiniSchemeB thus gives us constants and pre-bound symbols. This isn't very exciting, but it is a start.

Before we go on, there is a step that will make testing your work easier. Our workflow is first (define T (parse expression)) and then (eval-exp T init-env).

To simplify this, we give you a link to a file called REP.rkt. Download this into the same directory as env.rkt, parse.rkt, and interp.rkt. Open up REP.rkt and make it require your three files:

```
(require "env.rkt")
(require "parse.rkt")
(require "interp.rkt")
```

Change your MiniScheme.rkt file so that it requires REP.rkt and give it one line for its body:

```
(read-eval-print)
```

If you run the MiniScheme file it will pop up a textbox into which you can type expressions. Anything you type will be given to the parser and its output will be sent to eval-exp with the environment init-env. For example, if you type number 55 into the textbox it will reply 55. If you type x (no need for quotes) into the textbox it will give you the value init-env binds to x. As MiniScheme expressions get more complicated this will be a big time-saver.

For this to work your top-level objects must use exactly the names REP expects: parse, eval-exp, and init-env. With anything else (such as your datatypes) you can use any names you please.

You can now see how Labs 6 and 7 go. Each step adds one new kind of expression. You extend the parser to to build a parse tree for the expression, then you extend eval-exp to evaluate this parse tree. The grammar is recursive, as are parse and eval-exp, so you can mix up sub-expressions to produce very elaborate expressions.