Interpreter Project, Part 3[*]

Due November 21st at 10pm

There are two steps to submitting this assignment:

(1) Place your write-up in a Google Drive folder that is shared with me.

(2) Fill out the Interpreter Part 3 Submission Form.

1 Introduction to Procedures

This is a very important section: be sure you understand it before moving to the next section. This week, we are going to add *lambda* and *let* expressions to our interpreter. As we will see in the section for handling *let*, we can apply a simple syntactic transformation to let expressions that turn them into lambda expressions. In other words, once our interpreter has the ability to handle lambda expressions, we will have also written nearly all the necessary code to handle let expressions, too.

While you work on this portion of the interpreter, it is important to keep straight the definition of a procedure, the evaluation of a procedure, and the application of a procedure.

Introducing lambda expressions into our interpreter will add complications which we have not had to worry about up to this point. First, let's consider an easy example from Racket:

```
(define x 200)
(define y 100)

(define f
  (lambda (x)
     (+ x y)))

(f 50) ; returns 150
x      ; still 200
```

While it should seem obvious from the above example that the answers Racket returns are correct, there are some important details to notice. First, both x and y are defined in the environment before f is defined. When we apply the procedure f to the argument 50, we assign the local parameter x to be 50 and then evaluate *(+ x y)*. When we evaluate y, we find that it is bound to 100; when we evaluate x, we find that it is bound to 50; and so we return 150. However, when we exit the procedure and evaluate x by itself, we get 200, not 50. Think how we might do this and then consider the following code:

```
(define x 200)  ;same as above
(define y 100)  ;same as above

(define f       ;same as above
  (lambda (x)
     (+ x y)))
```

[*]Thanks to Rich Wicentowski for developing the original version of this project.

```
(define g
  (lambda (y)
     (f y)))

(g 50) ; ???
y      ; still 100
```

Now things have become more interesting. Just before we apply the procedure g to 50, x and y are still bound to 200 and 100, respectively. When we apply g to 50, the local variable y in g gets bound to 50, so *(g 50)* evaluates to *(f 50)* in an environment where x is 200 and y is now 50. When we apply f to 50, the local variable x gets bound to 50. So, what does *(+ x y)* evaluate to?

You'll find that Racket evaluates *(+ x y)* to be 150. This may seem odd. Why? Well, we know that the local variable x will evaluate to 50, leaving us only to evaluate y. So, what is y? Notice that there are two plausible answers here. If you thought that since y was bound to 50 in g, y should now be 50 in f, then you thought that Racket uses dynamic scoping. Although some implementations of Lisp, the language that Racket was derived from, use dynamic scoping, Racket does not.

If you thought that *(+ x y)* should be 150, then you thought that Racket uses static scoping, and you would be correct.[1] Static scoping says that the environment where you define a procedure should be the environment in which you apply the procedure. Let's see what that means by taking a look at how we will implement it.

The way that static scoping is implemented is through the creation of a **closure**. When a procedure, such as *(lambda (x) (+ x y))*, is defined, we will create a closure which will contain the actual definition of the procedure, as well as the environment in which it is defined.

When we go to define the procedure f, the environment contains a single frame with two bindings. (For this example, we will exclude the primitives and use cons pairs instead of mcons pairs to make it easier to read.) So, when we define f, the environment would be: *(((y 100) (x 200)))*. The closure we create will have both the definition of the procedure, and this environment. To help us differentiate between closures and primitives in our interpreter, we will tag the list with the symbol closure. So, the closure we create for this procedure will look like this:

```
(closure (lambda (x) (+ x y)) (((y 100) (x 200))))
```

At the start of this portion of the interpreter, I said that it was important to keep the definition, evaluation and application of lambda expressions separate. The code *(lambda (x) (+ x y))* is the definition of a procedure which evaluates to be a closure (such as the one above).

If we define a variable to be a procedure, as we do with f when we say *(define f (lambda (x) (+ x y)))*, then we simply add a new binding to our environment which binds f to the closure. (If you go poking around in the details, you'll find that this gives us an odd-looking recursively defined environment because the variable f is stored in the environment as being bound to a closure which itself stores the environment.)

[1]Most modern programming languages, for example C, C++, Java, and Racket, all use static scoping.

So, what happens when you type *(f 50)*? First, Racket evaluates both f and 50: Racket looks up f in the environment, finding a closure, and 50 evaluates to itself. Then we apply the closure to 50.

To apply a closure to a list of arguments, we do the following: First, the environment which is saved in the closure will get temporarily extended to include bindings of each of the procedure's parameters (which is just x in this example) to the values of the arguments (which is 50 here). Then, the lambda expression stored in the closure is evaluated with respect to this extended environment, and the resulting answer will serve as the final answer for the procedure application. So, after we extend the environment *(((y 100) (x 200)))* to include the new binding *(x 50)*, we evaluate *(+ x y)* and get 150.

We'll go into details in the next sections.

2 Defining and evaluating *lambda*

As mentioned in the introduction, when we define a lambda expression, we will construct a closure which is a tagged list containing the symbol closure, followed by the lambda expression, followed by the current environment:

```
(define make-closure
   (lambda (lambda-exp env)
     (list 'closure lambda-exp env)))
```

Here is an example of what happens in Racket when we evaluate a lambda expression:

```
> (lambda (y) (+ x y))
#<procedure>
```

What Racket is actually storing is a closure, though it hides the internal representation from you and reports #<procedure>. When we evaluate a procedure in our interpreter, we get the following:

```
INTERPRETER> (lambda (y) (+ x y))
(closure
   (lambda (y) (+ x y))
   (((empty ())
     (first (primitive first #<procedure first>))
     ...
    ))
)
```

Notice that it displays the entire environment, including all the primitive procedure definitions. This will get very tedious. I've provided you with a new print function, *i-print* which, along with a few other things, hides the environment when displaying closures.

When our interpreter encounters a lambda expression, it should create a closure. We can accomplish this by adding the following line to *i-eval*:

```
((lambda? exp) (make-closure exp env))
```

2.1 Testers and selectors for closures

Now we need to define testers and selectors for closures.

First, define the tester *lambda?*, which checks if an expression is a lambda expression, and the tester *closure?*, which checks if an expression is a closure.

```
(define lambda?
  (lambda (exp)
     ...))

(define closure?
  (lambda (exp)
     ...))

> (closure? (make-primitive '+ +))
#f
> (closure? (make-closure '(lambda (x) (+ x y)) global-env))
#t
```

Define the selector *procedure-parameters*, which returns the parameter names from a closure.

```
(define procedure-parameters
  (lambda (closure)
    ...))

> (procedure-parameters (make-closure '(lambda (x z) (+ x y))
                                       global-env))
(x z)
> (procedure-parameters (make-closure '(lambda () (display 5) (newline))
                                      global-env)
()
```

Define the selector *procedure-body*, which returns the body of the procedure from a closure.

```
(define procedure-body
  (lambda (closure)
    ...))

> (procedure-body (make-closure '(lambda (x z) (+ x y)) global-env))
((+ x y))
> (procedure-body (make-closure '(lambda () (display 5) (newline))
                                global-env))
((display 5) (newline))
```

Define the selector *procedure-env*, which returns the environment from a closure.

```
(define procedure-env
  (lambda (closure)
     ...))

> (procedure-env (make-closure '(lambda () (display 5) (newline))
                                (empty-env)))
()   ; this is the empty-environment
> (procedure-env (make-closure '(lambda () (display 5) (newline))
                                global-env))
...  ; returns the global environment
```

3 Applying lambda

All we have to do now is apply our lambda expressions (which are now closures) to a list of arguments. Let's follow this next example step by step. This expression should return 20.

```
((lambda (x) (+ x 30)) (/ -50 5))
```

First, *i-eval* will recognize this expression as a procedure application (since it's a list and you wrote *application?* in Part 2). It will then (recursively) evaluate the operator *(lambda (x) (+ x 30))* using the current environment, evaluate the list of operands (*(/ -50 5)*) using the current environment, and then call *i-apply* to apply the resulting closure to the list (-10). The closure obtained by evaluating the operator will look like this:

```
(closure (lambda (x) (+ x 30)) (((empty ())
                                 (first (primitive...)) ...)))
```

In *i-apply*, we will need to add a case to handle closures:

```
((closure? proc) (apply-closure proc vals))
```

The new procedure *apply-closure* should implement the steps described above. In particular, *apply-closure* should first extend the environment stored in the closure using the procedure's parameters bound to the list of values. Then it should evaluate the body of the procedure in the context of that new environment. In this case, a new frame containing the single binding *(x -10)* will be added to the procedure's environment, and then the sequence of expressions in the body (*(+ x 30)*) will be evaluated in that new environment, yielding a final result of 20.

Remember that the body of a procedure is a list of expressions to be evaluated, so you cannot use *i-eval* directly on the body. Instead, you must call *i-eval* on each expression, in order from first to last. You should be able to handle the evaluation of the procedure body using the *eval-begin* procedure you wrote in Part 2.

Make sure you clearly understand these steps before you try to update *i-apply* and write *apply-closure*. Once you've written your *apply-closure* function and have tested it thoroughly, you can uncomment the line beginning with *((closure? proc)* in *i-eval* and in *i-print*.

4 Implementing *let*

A *let* expression has the following syntax:

```
(let ((var1 val1) (var2 val2) ... (varn valn))
  exp1
  exp2
   ...
  expm)
```

As we have seen previously in class, this is simply shorthand for the following:

```
((lambda (var1 var2 ... varn) exp1 exp2 ... expm)
    val1 val2 ... valn)
```

Modify your interpreter so that it can recognize and handle *let* expressions. You should write procedures called *let?* and *let->lambda*. The procedure *let->lambda* takes a let expression and returns an equivalent application expression as specified by the above syntactic equivalence rule.

When *i-eval* encounters a *let* expression, all it needs to do is transform the expression into an equivalent application expression *(using let->lambda)*, and then just call *i-eval* again on the new expression.

At this point, you should be able to run nearly all of your code from your first few homework assignments. There are some special language constructs that will not work automatically: the built-in higher-order functions map, filter, and fold; and match.

5 Extra Credit: Higher-order functions

Racket's primitive higher-order functions require that their first argument is a Racket procedure (which Racket displays as something like #<procedure>). However, when our interpreter sees a procedure, it turns it into a closure, not a Racket procedure. For this reason, unlike with other primitives, we cannot directly use Racket's *map*, *filter*, and *fold* primitive.

5.1 Implementing *map*

Implement *map* in your interpreter. Only worry about the basic version of *map* which takes two arguments, a procedure and a single list, and provides as output a new list where each element is the result of applying the procedure to each item in the list. For example:

```
INTERPRETER> (map (lambda (n) (+ n 1)) '(1 2 3 4 5))
(2 3 4 5 6)
```

Note: While you might not care about the difference, doing what we just did for *map* means that in our interpreter, *map* is a special form, whereas in Racket, it is a primitive.

5.2 Implementing *filter*

Implement *filter* in your interpreter. Only worry about the version that takes 1 list and a procedure.

5.3 Implementing *fold*

Implement *filter* in your interpreter. Only worry about the version that takes exactly 1 list, an initial value, and a procedure. You may choose to implement either *foldl* or *foldr*.