CPSC 3740 Project Report

Steven Deutekom

Introduction

The purpose of this document is to describe my implementation for the 3740 interpreter project. I am going to do this in a slightly different way than is asked for in the project specifications. However, I will touch on things like project organization and data structure in each of the sections. The first will explain how I dealt with l*ambda*. This is because how I dealt with *lambda* affects the structure and organization of all the other elements of my interpreter. After this the report will follow the layout of the *startEval.rkt* file. Dealing with evaluation, environment and bindings, simple procedures, and finally *let* and *letrec*. Each section will discuss the program elements and also serve as extra documentation.

Lambda

The most important structural element of my interpreter is *lambda*. I used rackets *lambda* to create a procedure that can be returned and applied to a list of arguments and an environment. I decided that it would be easier to let the language create my closure for me rather than doing it myself.

When a *lambda* expression is parsed my interpreter takes the parameter list, lambda body, and the current environment and returns a procedure. This procedure takes 2 arguments. The first is a list of arguments that were passed to the *lambda* in the program that is being evaluated. The second is the environment that is current when this procedure is called.

When this procedure is run it binds all arguments, from the list of arguments passed to it, in order to the list of parameters from the *lambda* declaration. When this is done all arguments are evaluated using the environment that is passed to the procedure. Thus these arguments are evaluated with the environment that is current to the procedure call. Once this new list of parameters and their values is created it is appended to the front of the environment that was current when the *lambda* was created. Finally, the body is evaluated with this new environment and the result returned. This allows all variables that are in the *lambda* declaration to use the correct values. If the same variables were re-declared they won't be overridden.

Because of this structure all other functions for evaluating racket expressions in my interpreter were set up to have this structure. This drastically reduces the size of my evaluate function as it does not have to bother checking for each accepted operator.

Evaluation

The evaluation process of my interpreter is relatively simple. The *startEval* function simply creates a base environment and the program given to it to *myEval*. The starting environment is described in the next section, but it is a list of symbol procedure pairs containing all the keywords that the interpreter manages and functions that handle their evaluation. These procedures all follow the same style of interface as lambda.

The first thing *myEval* does is to check to see if the data passed to it is a single element. If it is then there are only two cases to consider.

1. The item is a symbol
    - If this is the case the interpreter assumes that it has a variable.
    - It looks it up and if there is no associated value an error is thrown. This was mostly to make sure that my interpreter behaved as racket does. However, it also allowed me to narrow down errors in my code by telling me whether the problem was passing an unbound variable to the racket interpreter or that I was not binding it properly.
    - If it has an associated value that value is returned.
2. The item is data of some kind
    - In this case the value is returned

If it is found that the expression passed to *myEval* is a pair the expression is a function. There are a few different possibilities in this case.

1. The first element of the function is a list
    - Since the first element of a valid racket function is always a procedure we assume that the evaluation of the first element will return a procedure. So we can evaluate it with the current environment and call the resulting procedure on a list of the arguments that followed it along with the current environment.
2. The first element of the expression is an actual procedure
    - Similar to above, all that needs to be done is to call the procedure on a list of its arguments along with the current environment. Then return the result.
3. If the first element is not an expression or a procedure then it must be a symbol
    - Here the variables value will be looked up in the current environment. Because (as will be discussed below) we cannot really be certain what is stored here, the value is substituted back onto the expression in place of the symbol and it is evaluated again. It will be processed by one of the above cases or if it makes it to the final case throw an error.
4. Finally, throw an error
    - Because the first element of a racket program must be a procedure if we get to this point whatever the first element of the current expression is not a procedure. All built in values are stored in the environment as procedures. So the first part of any function that is being evaluated can only be an expression, a procedure, or a symbol. If any of those do not actually evaluate to a procedure then this is an error.
    - Again, I throw my own error so I can see that the problem was that I passed something that was not handled properly by *myEval*. This of course assumes that all my tests pass valid racket programs to the interpreter.

This evaluation process is only possible because the environment stores the code to evaluate each racket construct in the environment. Then when a symbol is encountered we do not have to decide which one it is and then write code on it. Instead we can just look up its value and act appropriately. Exactly how this is done is detailed in the next section.

Environment

The basic environment for my interpreter uses a list of tuples. Each binding has a symbol as the first element of the tuple and a value as the second. This value can be a data value, a procedure, or even a racket expression depending on the reason that it was added. In all places where it is necessary new bindings are added onto the front (the *car* side).

To look up variable values I have a simple function that just looks at the first element of the

environment passed to it. If the binding contains the variable then the value is returned. Otherwise, the rest of the environment is recursively checked. If the variable is not found then a special constant *UNBOUND* is returned to signal the variable has no binding. This is generated a program start with the function *gensym*. This function just ensures that the value of *UNBOUND* is always unique so it can't clash with any other value in the program.

As discussed above my *lambda* results in procedures that are take a list of arguments and an environment. I decided to take this structure and use it to store the built in racket procedures (like '+, 'equal?, Etc.) in my base environment. The way that this works is that I have 3 functions that wrap a procedure so that it takes a list of arguments and an environment. These work for procedures that take 1, 2, or 3 arguments. For example, '+ is a binary operator. When I bind the *plus* procedure from racket to '+ and add it to the environment I first pass the procedure to *binaryOp*. This returns a procedure that calls plus on the first 2 elements of a list after evaluating these arguments with *myEval* and the current environment.

I found this really useful because it allows the addition of any racket procedure to my interpreter with one short line of code. With the caveat that the procedure must take 1, 2, or 3 arguments all the time and that none of those arguments are themselves procedures. I have a version of the interpreter that also has a number of string and math functions. The only code is added to the *builtin* function binding the result of wrapping the procedure to its symbol. I also wrote all my functions that handle different kinds of racket expressions that cannot use the wrappers with this signature. This way they can all be stored in the same way in the environment and I have to handle fewer cases in *myEval*.

I also defined a few small helper functions to make *lambda*, *let*, and *letrec* easier to implement. First there is a function *bind* that takes 2 lists. This returns a list of tuples where the elements of the first list are used as the symbols and the elements of the second list are used as the values. In *lambda* this is used in the created procedure to connect the arguments to the parameters before they are added to the environment. Second there is a function *eval-values* that takes a list of tuples. It uses *map* to return a list of the tuples after evaluating the second elements.

Simple Procedures

There are a few simple procedures that cannot be wrapped with my operator functions.

The first one is *if*. Racket does not allow it to be passed around like other procedures. However, it is just a function that takes 3 arguments. The condition and two different bodies. So I just evaluate the first 3 arguments of the list passed in, with the current environment, and call if on them.

The second one is quote. I found that using it as a procedure because I have to pull apart an expression and pass the *cdr* to *quote* that I had to do something different. I wonder if other people did the same thing. To get the actual value of the content passed to it I called *unquote* on the variable and then called *quasiquote* on the result. This has the effect of getting the value of a variable or expression and then quoting it. Though I am not exactly sure how it works, it does, until you break it somehow ;)

Let/Letrec

Finally, we have *let* and *letrec*. These functions turned out to not be too hard to work with once I switched my interpreter to work with the list of tuples environment.

Let takes the list of bindings given to it and evaluates all the values of those bindings using the previously mentioned *eval-values*. Then the result of this is added to the environment and the new environment is passed to *myEval* along with the let body. Now when the body is evaluated any variables that override previously declared ones will be found first. In order to maintain the proper functionality of *let,* the values of the bindings are evaluated before any of them are added to the environment. This way if a variable that is declared in both an outer and the current *let* level is used the proper scope will be maintained.

Letrec takes the *let* process and modifies it slightly. Instead of evaluating the values of the bindings we just add the list to the environment and evaluate the body. This delays the evaluation of the variables and allows them to be referenced recursively. Instead of doing what you mentioned in class having a constant to show a variable is uninitialized I just wait until the expression is in the environment to evaluate it. In the Evaluation section I talked about how we don't really know what kind of information that is in a variable when we have a variable as the procedure to a function. This *letrec* behavior is the reason I just reconstruct the function with the variables value and re-evaluate it. If the value of the variable is a variable or a racket expression it will just keep being replaced until the base case is reached and another value is used. This means that a deeply recursive function will build up a large nested list of the function code over and over again until the base case value is used and it unwinds evaluating everything with proper values. The funny thing is that while the process is a little harder to come up with it requires less work than let in some ways.

Program organization

I worked on this for a long time (as you know). I have worked very hard to break the whole project into logical chunks. Hopefully, this is relatively clear from the structure of this document, but I will say a little more about it. I tried to come up with a way to not have a large evaluation function that as things were added to the interpreter required adding more and more lines to this function. In the end this lead to adding the built in functions to the base environment. To me this makes the program more readable and more easily extendable. It makes each piece of the project nicely self contained. It actually adds a little to the size of the code compared to some of the code I have seen. However, I don't think line count is as important as being able to understand the code. I suppose that some of the decisions that I made can be thought of more as software engineering things rather than programming language things, but they are still helpful. On the programming language side of things I believe I did manage with my approach to use rackets built in structures to my advantage. I wrote less code for each rule for simple racket constructs because I was able to wrap them. This meant only writing code where I apply a procedure to evaluated arguments a few times rather than in every rule. Racket took care of my closures for me and I didn't have to worry about what to do with a closure when I got it in *myEval* because it was just in a procedure that could be called.

Testing

I wrote many tests. Following are 5 of them. Just the racket program and the result are included. I wrote a testing function that ran racket programs through both the racket interpreter and my interpreter and compared the result. If it failed then it would show me the expected and actual results and I could use those to work through my code. I got this idea from Soraj and Ethan. I also did a lot of simple tests, but here are a few of the more complex ones.

1. This test is just to make sure that let properly and ensure that it deals with bindings properly.

Code:
```
(let ([x 4]
      [y 10])
   (let ([y 5]
         [x y])
      (+ x y)))
```

Result:
```
15
```

2. In this test we declare a function that takes a function and a value as parameters and save it to a variable in a let. Then that variable is returned from the let. The result of the let is applied to 2 arguments.

Code:
```
((let ([a (lambda (f y) (f y))]) a) pair? '())
```

Result:
```
#t
```

3. Here we take the last test and make it better. The procedure stored in the let is used as in the body of another procedure that is returned from the let. This time the procedure returned from the let takes a procedure and passes it and an argument to the originally stored lambda.

Code:
```
(let ([a (lambda (f y) (f y))]) (lambda (x) (a x '()))) pair?)
```

Result:
```
#t
```

4. This test is an answer to programming question 2 for the second assignment. Finding the height of a tree. It is my algorithm before looking at your solutions so it is a little more work, but that is fine for testing. Doesn't have to be good, just valid racket.

Code:
```
(letrec ([greater (lambda (a b)
                    (if (> a b) a b))]
         [treeH (lambda (h x)
                  (if (equal? x (quote ()))
                     h
                     (childrenH (+ h 1) (cdr x))))]
         [childrenH (lambda (h lx)
                      (if (equal? lx (quote ()))
                         h
                         (greater
                            (treeH h (car lx))
                            (childrenH h (cdr lx))))))])
     (treeH
             0
             (quote (1
                      (2 (3 () ()) (4 () ((6 ()))))
                      (15 () ())
                      (16 () ())))))
```

Result:
```
4
```

5. This is the first programming question from assignment 2. It takes a tree and returns a list of all elements at a given level. Again this is not the best implementation.

Code:
```
(letrec ([levRec (lambda (n x cur res)
                      (if (equal? x (quote ()))
                         res
                         (if (= n cur)
                            (cons (car x) res)
                            (levList n (cdr x) (+ cur 1) res))))]
         [levList (lambda (n lx cur res)
                      (if (equal? lx (quote ()))
                         res
                         (levRec n (car lx) cur (levList n (cdr lx) cur res))))])
     (levRec
            3
            (quote (1 (2 (5 () ()) (6 ())) (3 ()) (4 ())))
            1
            (quote ())))
```

Result:
```
(5 6)
```

Final Thoughts

As for bugs and limitations, I believe that I have everything I need working without any bugs. I have passed all my tests and some other students tests and your tests. The only thing that I know is not perfect is that all symbols are considered variables. This makes it impossible to use them in any other fashion in a racket program. However, this has no effect on our implementation.

And when you inevitably find all kinds of code that breaks my interpreter please let me know what it is. I would love to see if I can get the full functionality desired at some point.

Sorry if the document is a little cumbersome to read. I waited till the last day to fully write it up even though I had 2 months to work on it.

Thanks for an interesting class and project!