Racket Interpreter – CPSC 3740 Final Project

Nathan Tipper & Vincent Cote

April 1st, 2019

For this project, we were tasked with creating a Racket interpreter using the racket language. To use the interpreter, the user calls startEval accompanied by a Racket program. This report will highlight the organization of the interpreter, data structures used, bugs or issues we encountered, and discuss how we tested our program.

# Organization

To implement our Racket interpreter we decided to use a list to keep track of the variables, in order to use this list we needed to add a couple helper functions. These helper functions are addBinding and findBinding, we also made the startEval function call myEval with the given program as well as this initial list of bindings. The “brain” of this interpreter is the myEval function, this function handles the parsing of the given program and calls the appropriate function based on what the program is asking for. To handle the lambda functions we also created a helper function to extract the variables and add them to our list.

# Data Structures

In order to keep track of our variables we decided to use a list, constantly adding new variables as needed. We decided to add the variables to the beginning of the list so that when searching through the list the first result we get is the one in scope. To search through the list we are simply using a linear search iterating from the beginning of the list to the end.

# Bugs and Limitations

To begin this project we started by attacking the more simple operations like arithmetic expression and comparison expressions. This stage went relatively well and we quickly had these operations defined as well as list traversals (car, cdr, etc.). We then moved on to let and lambda, at first glance we got these working early on, we used the following test codes for each:

(startEval '((lambda (x y z) (+ x z)) 1 3 5)) ; lambda expression - Expected Output: 6

(startEval '(let ([x 5]) (+ x 5))) ; let binding - Expected Output: 10

As these cases are simple implementation of let and lambda we had these working well and led us to assume we had let and lambda done, only letrec was left with time to spare. It turns out we where wrong, when testing let with lambda like:

(startEval '(let ([f (lambda (x) (\* x x))]) (f 5))) ; let-lambda binding - Expected Output: 25

We quickly realized our interpreter could not account for these kinds of scenarios. We started testing out some of the test code provided which led us to realize we were much further from the goal as expected with time running out. In many cases “fixing” on aspect would end up breaking another. In the end we managed to get some variation of let, lambda, and letrec to work but have found some bugs when trying more complex expressions. See the following list for known bugs:

1. Passing in a ‘quote’ as a value for a let function – eg:

(startEval '((lambda (x) (+ x (quote 1)))(quote 5)))

This bug was found when running some of the provided test code. Changing the ‘(quote 5)’ to simply 5 runs properly.

1. Passing in multiple variables inside a letrec function does not work – eg:

(startEval '(letrec ((x 5)(y 6))(+ x y)))

1. Nesting multiple lambda expressions does not work – eg:

(startEval '(((lambda (x) (lambda (y) (+ x y))) 1) 2))

1. Using car and cdr within lambda expressions does not work – eg:

(startEval '((lambda(x)(cdr x))'(1 2 3)))

1. We have scoping issues as found by one of the provided test code –eg:

(startEval ‘(let ((y 10)) (let ((f (lambda (x) (+ x y)))) (let ((y 100)) (f 2)))))

Running this uses the wrong value of y and returns 102

# Testing

For testing we started by adding simple test cases at the bottom of the file when we completed a new operation evaluation, this list of test grew as we added more functionality. As mentiond in the bugs and issues section, this caused us some problems as we did not test enough different cases for each operation. Our final list of these small test we ran was the following:

(startEval '()) ; Empty - Expected Output: 0

(startEval '(2)) ; Number - Expected Output: 2

(startEval '((define x 5) (define y (+ 3 4)) (\* x y))) ; Variables - Expected Output: 35

(startEval '(quote 4)) ; Quote - Expected Output: 4

(startEval '(+ (+ 7 3) (+ 2 3))) ; Addition - Expected Output: 15

(startEval '(- (- 7 3) (- 2 3))) ; Subtraction - Expected Output: -1

(startEval '(\* (\* 7 3) (\* 2 3))) ; Multiplication - Expected Output: 105

(startEval '(/ (/ 30 3) (/ 25 5))) ; Division - Expected Output: 2

(startEval '(= (+ 3 3) 5)) ; Equality - Expected Output: False

(startEval '(= (/ 20 5) 4)) ; Equality - Expected Output: True

(startEval '(> 6 (\* 3 2))) ; Greater Than - Expected Output: False

(startEval '(> (\* 5 4) 5)) ; Greater Than - Expected Output: True

(startEval '(< 20 (\* 5 4))) ; Greater Than - Expected Output: False

(startEval '(< (\* 5 3) 20)) ; Greater Than - Expected Output: True

(startEval '(>= 7 (\* 3 3))) ; Greatern Than - Expected Output: False

(startEval '(>= 9 (\* 3 3))) ; Greater Than - Expected Output: True

(startEval '(<= 10 (\* 3 3))) ; Greater Than - Expected Output: False

(startEval '(<= 9 (\* 3 3))) ; Greater Than - Expected Output: True

(startEval '((define y 9) (equal? y 8))) ; Object Equality - Expected Output: False

(startEval '((define x 5) (equal? x 5))) ; Object Equality - Expected Output: True

(startEval '(if (= (\* 7 3) (+ 15 7)) (+ 5 4) (\* 7 8))) ; if Condition - Expected Output: 56

(startEval '(if (= (\* 7 3) (+ 15 6)) (+ 5 4) (\* 7 8))) ; if Condition - Expected Output: 9

(startEval '((lambda (x y z) (+ x z)) 1 3 5)) ; lambda expression - Expected Output: 6

(startEval '(let ([x 5]) (+ x 5))) ; let binding - Expected Output: 10

(startEval '(let ([f (lambda (x) (\* x x))]) (f 5))) ; let-lambda binding - Expected Output: 25

(startEval '(letrec ([f (lambda (x) (if (= x 1) 1

(\* x (f (- x 1)))))])

(f 10))) ; letrec binding - Expected Output: 3628800

We also ran test cases in line after running the program, this is also how we ran through the provided test code.