CPSC 3740: Programming Languages
Final Project

(Spring 2019)

Davison Mathew, Zach Nelson,
Taranjot Kaur

Distribution Of Work

For our project, we divided the work up as evenly as possible and each worked on our separate parts before bringing them all together. We had group meetings to discuss and test our ideas on how to implement Lambda, Let, and Letrec. Our work assignments were as follows:

- Zach Nelson's portion was working with constants and variables, standardizing documentation, lambda expression and applying lambda expression to arguments.
- Davison Mathew's part was to work with lists (car cdr), conditionals, translating let and letrec into lambda expressions that have local binding using a "UNDEF" value, edits on the report.
- Taranjot Kaur's part was to work on evaluating and testing lambda expressions, local bindings and basic operations, and constructing the report.

Proceedings of the Program

We implemented an interpreter for a subset of Racket inside Racket. The main goal of our project was the implementation of the function startEval which takes list1 into consideration and apply various constructs like constants and variables , arithmetic operations(eg:- +,-,*,/) , relational operations(eg:- =,<=,>=,<,>) , conditionals(eg:- if), list operations(eg:- car,cdr,cons,pair?) , lambda expressions and applying lambda expression to an argument and local binding(eg:- let, letrec).

Key Data Structures

Lists were used in three key ways in the startEval program:

1. Evaluation List (named List1)
2. Values List (named List2)
3. Names List (named List3)

The Evaluation List is responsible for holding the list that needs to be evaluated. In this example, ((lambda (x) (+ x 1)) 3) the initial Evaluation list would be (lambda (x) (+ x 1)).
In that same example the Values List will be (3) and Names List will be (x).

Within this startEval2, we first check to see if the name and value lists are empty. If they are, then just return list1.
After this, it checks if list1 is equal to a variable name. If it is, if evaluates it further.
This further evaluation is done by checking the length of the list, if it is one, it will first remove any redundant brackets. Next, it goes through the list and checks what the first element within the list is, bringing it into the proper function based on what that element is. If it did not match any of those, first it echecks to see if the names and values are empty. If they are, return list1. If not, check for UNDEF from the letrec function. If it is not undef, then it should just be a letrec function name, and it starts the evaluation from that.

Implementation Functions

startEval -
    starting point for the program that does not require the Values or Names List
startEval2 -
    requiring a Values and Names List with the Evaluation list.
MyLambda -
    create a lambda expression and evaluate it.
MyLet -
    create an equivalent expression in lambda, so startEval2 can evaluate it.
MyLetAttributeNames -
    extract attribute names from let expression
MyLetAttributeValues -
    extract attribute values from let expression
MyLetrec -
    create an equivalent expression in lambda, so startEval2 can evaluate it.
MyLetrecAttributeNames -
    extract attribute names from letrec expression
MyLetrecAttributeValues -
    extract attribute names from letrec expression and UNDEF if needed
MyAdd -
    add two numeric values
MySub -
    subtract two numeric values
MyDiv -
    divide two numeric values
MyMult -
    multiply two numeric values
MyEqual -
    checks if two values are “equal?”
MyEqualSign -
    checks if two values are “=”
MyLessThanEqual -
    checks if two values are “<=”
MyGreaterThanEqual -
    checks if two values are “>=”
MyGreater -
    checks if two values are “>”
MyLesser -
    checks if two values are “<”
MyQuote -
    quote of the list
MyCar -
    car of a passed list
MyCdr -
    cdr of a passed list
MyCons -
    cons of two passed lists
MyPair -
    performs pair? on a list

Limitations

We were having problems with evaluating the left side of lambda expressions, but we were able to figure it out eventually. We needed to evaluate the param before evaluating the lambda expression. In the example, ((lambda (n) (* 2 n)) (+ 2 3)), our first implementation did not evaluate (+ 2 3) which lead to a problem later when the program tried (* 2 (+ 2 3)) in our add function. The error would be something like cannot * (+ 2 3) because it is not a number and the expected was (number?). This turned out to be a quick fix, by evaluating the values passed to the lambda expression before evaluating the whole lambda expression.

Currently we only have one problem with letrec where *(letrec ([fact (lambda (x y z) (if (= x 0) 1 (* x (fact (- x 1) y z))))]) (fact 6 7 8))* or any letrec param with multiple arguments for fact will result in an infinite loop. This is something we weren't sure was in the scope of the project as we were able to correctly evaluate the (*startEval '(letrec ([fact (lambda (x) (if (= x 0) 1 (* x (fact (- x 1))))]) (fact 10))).*

Test Cases

1. **Arithmetic Operations**
   - Addition
     (startEval '(+ 4 3)) :- This returns simple addition of two numbers and the answer will be 7.
   - Subtraction
     (startEval '(- 4 2)) :- This returns simple subtraction of two numbers and the answer will be 2.
   - Multiplication
     (startEval '(* 3 5)) :- This returns simple multiplication of two numbers and the answer will be 15.
   - Division
     (startEval '(/ 4 2)) :- This returns simple quotient of two numbers and the answer will be 2 .

2. **Relational Operations**
   - equal?
     (startEval '(yes yes)
     (equal? yes yes)
)

     This returns #t after checking the equality of the two arguments.
   - <=
     (startEval '(2 2)
     (<= 2 2))

     This returns #t because 2 is less than or equal to 2.
   - >=
     (startEval '(2 3)
     (>= 2 3))

     This returns #f because 2 is not greater than or equal to 3.
   - >
     (startEval '(2 3)
     (> 2 3))

     This returns #f because 2 is not greater than 3.
   - <
     (startEval '(2 3)
     (< 2 3))

     This returns #t because 2 is less than 3.

3. Lists
    - pair?

```
(startEval '(1 2)
  (pair? '(1 2)))
```

This returns #t because 1 and 2 form a pair of the list.

- car

```
(startEval '(1 2)
  (car '(1 2)))
```

This returns 1 because 1 is the first element of the list.

- cdr

```
(startEval '(1 2)
  (cdr '(1 2)))
```

This returns 2 because 2 is the second element of the list.

4. Lambda

```
startEval  (lambda (x y)
  (- y x)))
(subtract 4 9)
```

This returns 5 after the evaluation of lambda expression.

5. (let ([x 3][y 2]) (+ x y))
   This returns 5, as 3 + 2 should equal 5, and shows that you can have multiple parameters within the let function (X and Y).

6. Letrec

```
(letrec ((even?
        (lambda (n)
          (if (zero? n)
              #t
              (odd? (- n 1)))))
       (odd?
        (lambda (n)
          (if (zero? n)
              #f
              (even? (- n 1))))))
  (even? 10))
```

This returns #t after the evaluation of letrec.

7. Letrec

```
(letrec ((fact
          (lambda (x)
          (if (= x 0) (quote 1)
          (* x (fact (- x 1))))))
         (fact 10)))
```

This prints 3628800, as was used as an example in the project description. This proves that our program is able to evaluate letrec functions properly.