CPSC 3740 PROJECT WRITTEN REPORT

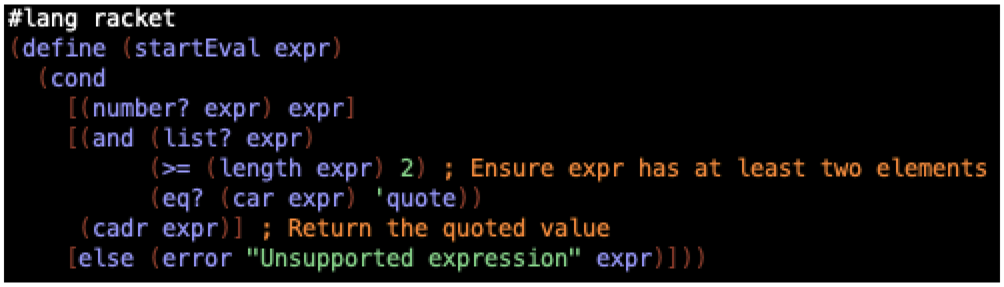
GROUP MEMBERS: PATRICK PAJDA, BEN HOLT, PEISON JIANG, SHTEGTAR QELAJ

How Did You Organize Your Program?

The program is organized around a central function, startEval, which evaluates Racket expressions recursively within a specified environment. The code is modular, with specific sections dedicated to handling different types of expressions and operations. Key components include:

1. **Core Evaluation Function (startEval):**
    a) Responsible for evaluating various types of expressions, such as arithmetic operations, conditionals, let, letrec, and function applications.
    b) Handles the environment (env) to maintain variable bindings and scope throughout the evaluation.
2. **Environment Management:**
    a) The environment is passed through recursive calls to manage variable bindings dynamically.
    b) lookup is used to retrieve values associated with variables in the current environment.
3. **Expression Handling:**
    a) Specific cases for each expression type, such as:
        - **Constants and Variables** (x, quotes, 5)
        - **Relational Operators** (<,>,<=,>=).
        - **Lists**
        - **Lambda expression**
        - **Arithmetic expressions** (+, -, *, /).
        - **Conditionals** (if).
        - **Bindings** (let and letrec).
        - **Function applications** using closures.
4. **Debugging:**
    a) Debugging statements were added during development to trace the evaluation process and identify issues. These statements were removed or commented out in the final version for clarity.

Starting Implementation



```
#lang racket
(define (startEval expr)
  (cond
    [(number? expr) expr]
    [(and (list? expr)
          (>= (length expr) 2) ; Ensure expr has at least two elements
          (eq? (car expr) 'quote))
     (cadr expr)] ; Return the quoted value
    [else (error "Unsupported expression" expr)]))
```

Functionalities:

a) Handling numbers: Numbers are returned as-is since they evaluate to themselves.
b) Handling quoted expressions: Explicit (quote…) expressions return the quoted value.
c) Unsupported expressions: Anything else raises an error.

Why Start Here

a) Numbers and quoted expressions are the simplest constructs in the language subset you're implementing.

From there we built the program block by block.

What Are the Key Data Structures Used and What Were They Used For?

1. **Environment (env):**
    a) A list of pairs (variable . value) that represents the current scope of variable bindings.
    b) Used to dynamically look up variables and extend scope for let and letrec.
2. **Closures:**
    a) Represented as a list: '(closure params body env), where:
        - params: The parameters of the function.
        - body: The body of the function.
        - env: The environment where the function was defined.
    b) Used to encapsulate functions and their environments for proper evaluation.
3. **Placeholders:**
    - Used in letrec as temporary values ('placeholder) during initial setup.
    - Allow recursive bindings to reference themselves while being defined.

Limitations/Bugs:

1. **Error Handling:**
    - The program currently has limited error handling for malformed expressions or edge cases, such as missing operands.

2. **Infinite Recursion:**
    - Misuse of letrec without a base case can lead to infinite loops.
3. **Division by Zero:**
    - There is no explicit check for division by zero, which can cause errors.
4. **Performance:**
    - For deeply nested or complex expressions, the program may experience stack overflow or performance issues due to heavy recursion.

How Did We Test Our Interpreter?

Testing focused on ensuring the correctness of different expression types and edge cases. Below are 5 sample test cases that demonstrate the interpreter's functionality:

Test 1: Arithmetic Operations

```
(startEval '(+ 2 3) '())       ; Expected: 5
(startEval '(* 4 5) '())       ; Expected: 20
(startEval '(- 10 3) '())      ; Expected: 7
(startEval '(+ x 3) '((x . 2))) ; Expected: 5
```

Explanation: Validates the handling of basic arithmetic operations and variable substitution.

Test 2: Conditional Expressions

```
(startEval '(if #t 42 0) '())           ; Expected: 42
(startEval '(if (= x 10) 1 0) '((x . 10))) ; Expected: 1
(startEval '(if (> x y) x y) '((x . 4) (y . 6))) ; Expected: 6
```

Explanation: Ensures that the interpreter correctly evaluates conditions and selects the appropriate branch.

Test 3: Recursive Functions with letrec

```
(startEval '(letrec ((factorial (lambda (n)
                          (if (= n 0) 1 (* n (factorial (- n 1)))))))
         (factorial 5)) '()) ; Expected: 120
```

Explanation: Demonstrates the correct handling of recursive definitions and function applications.

Test 4: List Operations

```
(startEval '(cons 1 '(2 3)) '()) ; Expected: '(1 2 3)
(startEval '(car '(4 5 6)) '())  ; Expected: 4
(startEval '(cdr '(7 8 9)) '())  ; Expected: '(8 9)
(startEval '(null? '()) '())     ; Expected: #t
```

Explanation: Verifies that basic list operations (cons, car, cdr, null?) work correctly.

Test 5: Function Application

```
(startEval '((lambda (x) (+ x 1)) 5) '()) ; Expected: 6
(startEval '((lambda (x y) (* x y)) 3 4) '()) ; Expected: 12
(startEval '((lambda (x) ((lambda (y) (+ x y)) 2)) 3) '()) ; Expected: 5
```

Explanation: Confirms that lambda functions and nested function applications behave as expected.

Conclusion

The program successfully implements a basic interpreter for a subset of Racket, handling arithmetic operations, conditionals, recursive functions, and list manipulations. While it functions correctly for the tested cases, it could benefit from improved error handling, performance optimizations, and additional testing for edge cases. This project demonstrates the core principles of interpreters and highlights the power of Racket for building recursive and functional programs.