Programming Languages Final Project Report

(Fall 2024)

Tarun Pokra

Chamod Jayathilake

Connor Pittman

Yunfan Yang

1. Project Execution

1.1 Roles and Responsibilities

To ensure an efficient workflow, we divide the tasks evenly among group members, with each person taking responsibility for specific parts of the project. After completing individual parts, we used GitHub to collaborate and integrate our work into a cohesive final product.

Tarun Pokra focused on functional tasks, specifically arithmetic operations and contributed to the report. He designed and implemented mathematical functions, ensuring proper integration and functionality within the project's broader computational framework.

Yunfan Yang worked on local bindings and conditionals, implementing local variables, managing scopes, and testing conditional logic to ensure correct functionality and decision-making processes within the program's execution.

Chamod Jayathilake involved working with lists, implementing operations, and contributing to the report by editing and structuring the content, ensuring list functions aligned with the project's functional goals.

Connor Pittman handled constants, variables, relational operations, lambda expressions, and function application, ensuring correct application of lambda functions, relational logic, and variable handling within the project's overall structure.

2. Objective and scope of the project

2.1 Objective

The objective of this project is to design and implement an interpreter for a subset of Racket, using Racket itself. The core function, startEval, takes a Racket program as an argument, evaluates it, and returns the result. The program assumes that the input Racket code is syntactically correct.

2.2 Scope

The interpreter must support constants and variables (numbers, variables, and quote), basic arithmetic operations (addition, subtraction, multiplication, division), relational operators (equal?, =, <, >, <=, >=), and list operations (car, cdr, cons, pair?). Additionally, it should handle conditionals, plain lambda expressions, function application, and local bindings (let and letrec).

3. Key Data Structures and Functions

Evaluation:

The program and base environment are created by the startEval function and passed to myEval. The following section describes the beginning environment, which is a collection of symbol procedure pairs that include every keyword the interpreter can use. And oversees and performs the tasks involved in their evaluation. The interface used by lambda is the same.
Upon receiving data, myEval first determines whether it is a single element. If so, there are just two situations to consider. So, assessment procedure can occur solely because the environment keeps the code needed to assess each racket built in the surroundings. When a symbol is met, we do not need to decide, identify which one it is and then create code for it. Instead, we can simply find its value and respond accordingly. For example, Lambda is our interpreter's most crucial structural component. Created a procedure that can be applied to an environment and a list of arguments using Rackets lambda. Our interpreter returns a procedure after parsing a lambda expression, taking the parameter list, lambda body, and current environment. This process requires two arguments. A list of arguments presented to the lambda in the program under evaluation is the first. The current environment at the time this operation is invoked is the second one. The list of parameters from the lambda declaration is bound to the list of arguments from the list of arguments provided to it when this procedure is executed. Following completion, the environment supplied to the procedure is used to assess each parameter. When the lambda is generated, the previously existing environment's front end is updated with this new list of parameters and their values. The body is then assessed in this altered setting, and the outcome is given back. This enables the proper values to be used by all variables contained in the lambda declaration. The variables would not be overridden if they were re-declared. Our interpreter's other racket expression evaluation routines were all configured to have this structure because of this one. Our evaluate function is significantly smaller.

Organization:

Basically, possess an extensive assessment function that as matters were included in the interpreter, necessitating the addition of increasingly more lines to this function. This resulted in to incorporate the built-in functions into the core environment. This made the program overall much more readable. We were also able to utilize the built-in features of rackets to benefit us. We created fewer lines of code for each rule to keep it simple.

4. Limitation and Bugs/ Known limitations

One of the known limitations we faced was that there is limitation is if we are missing functionality specifically. Especially we are to create and run function test cases.

4.2 Identified Bugs and Issues

There were some issues we faced where the readability of the code outputs became a little difficult especially for the test cases, so we added new lines each time. As well as calling certain files at the top, were not called properly before.

5. Testing and Validation

Test one example:

declare a function that takes a function and a value as parameters

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

Result: #t

Test 2 example:

procedure stored in the let is used as in the body for another procedure

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

Result: #t

// the same as the one before.

Test 3 example:

To make sure bindings are being done properly.

(let ([x 4]

[y 10])

(let ([y 5]

[x y])

(+ x y)))

Result: 16,

Correction, result is 15, because y 10, and let (y 5).

Test 4 example:

To figure out the height of a tree, note the code example here form our project is cut down and simplified.

(treeH

0

(quote (1

(2 (3 () ()) (4 () ((6 ()))))

(15 () ())

(16 () ())))))

Result: Four.

Test 5 Example:

Using a tree and returning a list for every single element there. Note again, we cut down our code for simplicity and for the sake of the example from our project code.

(levRec

3

(quote (1 (2 (5 () ()) (6 ())) (3 ()) (4 ())))

1

(quote ()))

Result: (5, 6), in an (x, y) format for readability.