University of Lethbridge

Course Project

Racket Interpreter in Racket

Gideon Richter

Programming Languages CPSC3740

Professor Howard Cheng

April 1st, 2019

Project Description

The goal of this project was to implement an interpreter for the functional Scheme based programming language Racket, written *in Racket*.

It implements a subset of functionality, including:

- Constants
- Arithmetic operators
- Relational operators
- Lists and list operations
- Condition statements
- Lambda expressions
- Local bindings with let

Program Organization

The code is organized as a list of functions. The only user-facing function is `startEval`, which takes a valid Racket expression as a parameter, such as `'(+ 2 2)`, and the result of the expression is returned. `startEval` is similar to Racket's own `eval` function, but it is not used within.

`startEval` is written as a single conditional statement, which looks at the first element of the given expression to determine what to do.

The trivial case, where the expression is a quote or a symbol defined in the environment, requires no evaluation.

```
> (startEval '(quote 5))
5
> (startEval '(let ((x 5)) x))
5
```

Figure 1: quote and symbol evaluation

If the expression is a conditional statement, the predicate is evaluated and either the first or second argument is evaluated based on the result.

```
> (startEval
   '(if (equal? 5 5)
        'yes
        'no))
'yes
> (startEval
   '(if (equal? 5 10)
        'yes
        'no))
'no
```

Figure 2: conditional statement evaluation

If the expression is a `let` we simply replace it with its lambda equivalent and then evaluate it as such. The recursive functionality of `letrec` is not implemented, and as a consequence, no recursive expressions may be evaluated with this interpreter.

```
> (convert-let
   '(let ((x (+ 5 10))) x))
'((lambda (x) x) (+ 5 10))
```

Figure 3: let conversion to lambda

Otherwise, any sub-expressions are recursively evaluated and passed as arguments to the given racket operator or user-defined function.

The functions are written in essentially the same order as they were needed. Every function defined has unit tests written in a test file ‘test_startEval.rkt’. Because of this, program correctness throughout development was easier to maintain and refactoring provably equivalent.

Key Data Structures

The data structures used are:

- list
- association list

A list and list operations are key to Racket (as in in Scheme, Lisp) and are used to build results and breakdown the given expression.

An association list is used to manage the environment as a list of local bindings. It looks like this `((a 5) (x (lambda (...))))`, where ‘a’ is associated with the value ‘5’ and ‘x’ is associated with the lambda expression `(lambda (…))`. The environment (or any association list) is accessed with the function `dict-get`, which executes a linear search of the list and returns the second element (the value) of a list where the first value matches a given key.

```
> (define env (dict-update 'key 'value '()))
> (dict-get 'key env)
'value
```

Figure 4: environment definition and access

Initially, this environment consists of only predefined racket operators as a matter of convenience to make function evaluation easier to understand. Otherwise in the eyes of the user, the initial environment is effectively empty.

When a user introduces a local binding with `let`, it is added to the environment using after evaluation of the given value by way of `dict-update`. If a symbol has already been defined by the same name, the new value is used, but because we return a new copy of the environment with each addition, nested environments remain constant over time.

Limitations and Bugs

- Racket's recursive local binding `letrec` is not implemented, and the result of `let` in a recursive fashion is undefined.
- The 'second form' of `let` provided in the Racket documentation is not implemented https://docs.racket-lang.org/reference/let.html (paragraph 2 of `let` heading).
- The only time we handle an exception is one 1) a user enters an unsupported unary/binary operator, such as `list?` and 2) a user tries to access a symbol that is not defined in the local environment. Because of this, exceptions are difficult to diagnose.

Testing

This project has primarily been tested using unit tests, with full coverage completed over 79 tests with 'test_startEval.rkt'. In addition, these tests include all test examples provided in the 'lib3740' directory that do not use `letrec`.