Carleton University
Department of Systems and Computer Engineering
SYSC 3101 - Programming Languages - Winter 2018

Assignment 2 - Racket Interpreter Design and Implementation

Posted: Monday, March 26, 2018

Due: Wednesday, April 11, 2018, 11:55 p.m.

Introduction

In this document, *Racket* refers to the language and interpreter provided by the DrRacket IDE, and *racket-1* refers to the Racket program that interprets a subset of Racket. You have been provided with two implementations of racket-1. File racket1.rkt contains the interpreter developed at UC Berkeley. File racket1-trace.rkt contains the same interpreter, but it has been modified slightly to display information that helps us trace its execution as it evaluates expressions.

Part 1

You do not have to submit your solutions for Part 1. These exercises will help you understand racket-1 before you do the programming exercises in Part 2.

Exercise 1

Launch DrRacket. In the Interactions area, type the expressions listed in the table. Note what Racket's REPL displays.

Expression	Racket displays	racket-1 displays
`5`		
`*`		
`(quote (1 2 3))`		
`(if (= 5 (+ 2 3)) "equal" "not equal")`		
`(lambda (x) (+ x 1))`		
`(* 2 3)`		
`(* 2 3 4)`		
`(* 2 (* 3 4))`		
`((lambda (x) (+ x 1)) 5)`		

Open racket1.rkt in DrRacket, and click Run. Start racket-1's REPL by typing `(racket-1)` in the Interactions area. Repeat the experiments you ran with Racket (type each expression after the `Racket-1:` prompt). To stop racket-1 and return to Racket, just evaluate an illegal expression; for example, `()`.

Were there any experiments where the values displayed by Racket differed from those displayed by racket-1?

Exercise 2

Take your time with this exercise. It is intended to help you understand how racket-1 evaluates different types of expressions. Adding new features to the interpreter should then be straightforward.

Open racket1-trace.rkt in DrRacket, and run racket-1's REPL. Type the same expressions you used in Exercise 1 and observe the execution traces. For each expression, how many times is `eval-1` called? How many times is `apply-1` called? How many times is `substitute` called? Make sure you understand each call to procedures `eval-1`, `apply-1` and `substitute`. What arguments are passed to each call? What does each procedure return?

Exercise 3

One of your colleagues has decided to reorder the `cond` clauses in `eval-1` so that the clause for procedure applications (`pair?`) appears after the clause for symbols but before the clause for quotes. The goal is to make the interpreter more efficient. The reasoning behind this decision is: because programs usually contain more applications than quotes, ifs and `lambda` definitions, the modified `eval-1` will usually check fewer clauses than the original `eval-1` before identifying the type of an expression.

What is wrong with this plan? (Hint: What will the modified evaluator do with the expressions `(quote (1 2 3))` and `(lambda (x) (+ x 1))`?)

Exercise 4

When executed by Racket, this expression uses `map` to apply a `lambda` procedure to each number in a list of integers. The result is a list containing the squares of the numbers in the list.

```
> (map (lambda (x) (* x x)) '(1 2 3 4))
'(1 4 9 16)
```

All of Racket's primitives are automatically available in racket-1, so you might think you could use Racket's primitive `map` function. Try this experiment in racket-1 (use the interpreter in racket1-trace.rkt, so that you have an execution trace to follow) :

```
Racket-1: (map (lambda (x) (* x x)) '(1 2 3 4))
```

Explain what happens.

Part 2

Modify racket1.rkt with your solutions to Exercises 5 through 7, and submit this file to cuLearn. Use comments to indicate clearly the locations of all your modifications. Consider using racket1-trace.rkt to prototype your solutions, adding `display` expressions to help you trace your modifications. After testing your solutions, copy them (without the `display` expressions) to racket1.rkt.

Exercise 5

Use Racket to evaluate these expressions. Note the values displayed by Racket's REPL.

```
(quote 1)

(quote (1 2 3))

(quote 1 2 3)
```

Now run racket-1, and evaluate the same three expressions. Clearly, the way that racket-1 evaluates `quote` expressions with multiple arguments is broken. Change the racket-1 interpreter in racket1.rkt so that it displays an error message and terminates when it evaluates a `quote` expression with multiple arguments.

Exercise 6

Write a `map-1` primitive for racket-1 (call it `map-1` instead of `map` so you and racket-1 don't get confused about which procedure is which). For example:

```
Racket-1: (map-1 (lambda (x) (* x x)) '(1 2 3 4))
'(1 4 9 16)
```

Hint 1: define a helper procedure named `map-exp?` that determines if an expression begins with `map-1`. `map-exp?` will be similar to the helper procedures in racket-1 that determine if an expression is a `quote`, `if` or `lambda` expression.

Hint 2: your `map-1` primitive should call Racket's `map` primitive to perform the mapping operation. As an example of how a racket-1 primitive can use a Racket primitive, have another look at how the `cond` clause that handles `if` expressions calls Racket's primitive `if` procedure.

Hint 3: you do not have to change `apply-1` or `substitute`. Defining `map-exp?` and adding a few lines of code to `eval-1`, is all that's required.

If you're stuck, review Part 1, Exercises 1 and 4.

Exercise 7

Racket's `and` form provides a way of combining tests. The syntax of this form is:

```
(and expr1 expr2 expr3 ...)
```

An `and` form produces `#f` if any of its expressions produces `#f`. Otherwise, it produces the value of its last expression.

If no expressions are provided, the result is `#t`; that is, `(and)` produces `#t`.

If a single expression is provided, `(and expr)`, the result of the `and` expression is the result of `expr`. Some examples:

```
> (and 1)
1
> (and (+ 2 3))
5
```

Otherwise, the first expression, `expr1`, is evaluated. If it produces `#f`, the result of the `and` expression is `#f`, and the remaining expressions are not evaluated. If evaluating `expr1` produces `#t`, the result of the `and` is the same as an `and` expression containing all the expressions except `expr`1. Some examples:

```
> (and #f (error "doesn't get here"))
#f

> (and #t 5)
5

> (and (= 5 (+ 2 4)) (* 3 4))
#f  ; (* 3 4) is not evaluated

> (and (= 5 (+ 2 3)) (* 3 4))
12

> (and (= 5 (+ 2 3)) (< 6 (* 4 3)))
#t

> (and (= 5 (+ 2 4)) (< 6 (* 4 3)))
#f  ; (< 6 (* 4 3)) is not evaluated
```

Modify the racket-1 interpreter to add the `and` special form. Ensure that as soon as a `#f` value is computed, your `and` returns `#f` without evaluating any further expressions.