Programming Language Theory

Evan Halloran

2025

These notes will cover the basic theory of programming languages, including the theory of computation, formal languages & syntax, and λ-calculus. We will work in Racket, a LISP (List Processing) language. Racket is a functional programming language, different from declarative Python and Java. Everything in Racket is either a variable, a procedure, or an application of a procedure. The motivation behind engineering programs in Racket will become clear when the λ-calculus is introduced. With this in place, we are able to build a bootstrapped interpreter that can read and evaluate any program in Racket we want.

1 Introduction

The easiest environment to code in Racket is DrRacket, an interpreter and IDE in one. DrRacket contains a Definitions and an Interactive window. The later is used much like a terminal interface for other programming languages, allowing the user to enter one-line prompts and receive outputs. We will exclusively use the Definitions window which enables us to write multi-line code and execute it as a program. It is important to understand however that our aim is not to build programs at all, but rather to understand the impetus behind programming language theory. Thus most of our early work in Racket will be entirely expository and focused only on the syntax of the language and functional programming.

In Racket, we can enter numbers into the Definitions window and have them displayed in the terminal at runtime.

```
42
0
-42
```

```
42
0
```

```
-42
```

We can use the keyword **define** to assign values to names (variables).

```
(define x 42)
```

Let's examine the syntax of this a little closer. Racket (following operator theory) is a prefix operator language, meaning a function call consists of the function's name first and its arguments second. A function call is always enclosed in parentheses, and the only things enclosed in parentheses in Racket are function calls. Thus, we can see that **define** is a function which takes in two arguments: x and 42. This function assigns the value of the second argument to the name of the first, like a variable declaration in other programming languages. We can then call this variable later and its value will be substituted:

```
x
```

```
42
```

Here is another example of a function in Racket, called **let**. It takes as its first argument a list of variable bindings. Its second argument is then evaluated in terms of the values of the variables. The variables cease to exist after the let function.

```
(let ((x 10)
      (y 32))
  (+ x y))
```

```
42
```

We see that x is bound to the value of 10 and y is bound to the value of 32. The second part of the **let** procedure then evaluates the function call $(+xy)$. In this procedure, the function under consideration is the built-in addition function "+". It takes in two arguments and returns their sum. The **let** procedure then returns the final value of the expression, 42.

After **let** has run, if we try to call x again we will get back 42 because that is the value that was globally bound to it by the **define** procedure. If we try to call y, we will get an error.

Here are some examples of the addition procedure.

```
(+ 10 32)
(+ 10 2)
(+ 10 0)
```

```
42
12
10
```

We can also create our own procedures for use. The general syntax is exhibited in the following function definition.

```
(define add10
  (lambda (n)
    (+ 10 n)))
```

We have seen **define** before, and it is doing entirely the same thing: it is binding the value of the second argument to the name of the first. That is, **add10** is now bound to the value of the second argument, which is a function. A lambda-function is a function without a name. The lambda function above takes as input a value n and returns the sum of 10 and n. This nameless function is assigned to **add10**, allowing us to call upon it later.

```
(add10 32)
```

```
42
```

Lambda functions can take in any finite amount of arguments:

```
(define add10-2
  (lambda (n n1)
    (+ 10 n)))

(define add10-3
  (lambda ()
    (+ 10 1)))

(add10-2 12 32)
(add10-3)
```

```
22
11
```

Racket also has Boolean values.

```
#t
#f
```

These will be useful when dealing with conditional execution.

```scheme
(cond
  (test1 value1)
  (test2 value2)
  ...
  (else valuen))
```

A cond block is a series of tests and corresponding values. Each test is an expression that must evaluate to a Boolean value. When the cond block is entered, test1 is evaluated. If test1 is true, then value1 is returned. If it is false, then test2 is evaluated. If test2 is true, then value2 is returned. If it is false, test3 is evaluated, and so on. If all tests fail, then valuen is returned.

```scheme
(cond
  (#t 12)
  (else 42))

(cond
  (#f 12)
  (else 42))

(cond
  (#f 12)
  (#t 32)
  (else 42))
```

```
12
42
32
```

We can use an if-statement to alternatively write conditionals.

```scheme
(if condition conseq alter)
```

When an if block is entered, the condition is evaluated. If the condition is true, the consequence is returned. If the condition is false, the alternative is returned.

```scheme
(if #f 12 42)
```

```
42
```

Racket has some other useful built-in functions. The **zero?** procedure is a one-place function that returns true if its argument is 0 and false otherwise.

```
(zero? 2)

(cond
  ((zero? 3) 12)
  (#t 32)
  (else 42))
```

```
#f
32
```

We can now start building some interesting functions from mathematics. Using **cond** and $<$, we can recreate the absolute value function.

```
(define
  (λ (n)
    (cond
      ((< n 0) (* -1 n))
      (else n))))

(abs -42)
```

```
42
```

We can equivalently make our absolute value procedure with an if-statement.

```
(define abs-if
  (λ (n)
    (if (< 0 n) (* -1 n) n)))
```

One of our main tasks in these notes will be to reduce some common place functions to their most primitive mathematical definitions. In most cases this will be achieved using natural recursion. A definition is naturally recursive (NR) if its recursive function call is wrapped inside another procedure. We will first investigate the addition function and provide a naturally recursive definition using only **zero?**, **add1**, and **sub1**:

```
(define add
  (λ (m n)
    (cond
        ((zero? m) n)
        (else (add1 (add (sub1 m) n))))))

(add 0 13)
(add 2 13)
(add 13 0)
```

```
13
15
13
```

The base case asks whether m=0; if yes, then n is returned as the sum. The recursive step reduces the problem of (+ m n) to the simpler problem of (add1 (+ (sub1 m) n)). This is how additive arithmetic is defined on nonnegative integers.

We can provide a similar definition for multiplication:

```
(define mult
  (λ (m n)
    (cond
      ((zero? m) 0)
      (else (add (mult (sub1 m) n) n)))))
```

Another type of value we will be working with is the pair. The **cons** function creates a pair. The **car** function (content of the address register) returns the first item in the pair, while the **cdr** function (contents of the decrement part of the register) returns the second item.

```
(define pair (cons 1 2))

(car pair)
(cdr pair)
```

```
1
2
```

We can nest pairs within each other.

```
(cons 1 (cons 2 (cons 3 '())))
(cons 1 (cons 2 3))
```

```
'(1 2 3)
'(1 2 . 3)
```

Another type of value is a list. The empty list is the value **null**, or equivalently **'()**.

```
(null? (cons 1 (cons 2 (cons 3 '()))))
(null? '())
```

```
#f
#t
```

However, non-empty lists are just pairs whose car is a member of the list and whose cdr is the rest of the list.

```
(car (cons 1 '()))
(cdr (cons 1 '()))
```

```
1
'()
```

Let's create a function that takes in a list l and a number n and returns the nth member of the list.

```
(define nth-memb
  (λ (l n)
    (cond
      ((null? l) (error "not found"))
      ((zero? n) (car l))
      (else (nth-memb (cdr l) (sub1 n))))))

(define l (cons 1 (cons 2 (cons 3 '()))))
(nth-memb l 0)
(nth-memb l 2)
```

```
1
3
```

Note that this is not a naturally recursive function, since there is no wrapper function around the recursive function application.

This is a function which removes all instances of 8 from a list.

```
(define rember8
  (λ (l)
    (cond
      ((null? l) '())
      ((eqv? (car l) 8) (rember8 (cdr l)))
      (else (cons (car l)
                  (rember8 (cdr l)))))))

(rember8 '())
(rember8 '(8 3))
(rember8 '(1 8 3))
(rember8 '(1 8 2 8))
```

```
'()
'(3)
'(1 3)
'(1 2)
```