Racket Programming Language

Seyed Ali Nedaaee Oskoee

2025 spring

فهرست

1- Introduction

1-1 What is the Racket Programming Language?

Racket is a general-purpose, multi-paradigm programming language. It belongs to the Lisp family of languages. Racket is mainly used for teaching programming concepts, developing new programming languages, and data processing.

1-2 Features of the Racket Programming Language

- **Multi-purpose:** Used for both educational purposes and the development of complex software systems.
- **Functional Programming:** Supports higher-order functions and functional programming paradigms.
- **Powerful Macro System:** Enables the creation of new languages through advanced macros.
- Supports **Object-Oriented** and **Procedural** Programming.
- Has a **dedicated IDE** called DrRacket.

1-3 Applications of the Racket Programming Language

- Teaching programming
- Developing new programming languages
- Data analysis and text processing
- Game and graphics development
- Automation scripting

2- Installing Racket

2-1 To install Racket on Linux systems, simply run the following commands in the terminal to install it via Snap**:**

```
$ sudo apt update
$ sudo apt install snapd
$ sudo snap install racket
```

2-2 To install Racket on macOS, you only need to run the following command:

```
$ brew install --cask racket
```

2-3 **To install Racket on Windows**, go to the official Racket website, choose the appropriate Windows version, download the .exe file, and follow the installation steps. Once the installation is complete, you will find DrRacket in your Start menu.

After installation, you can open the DrRacket application and run the following code to verify that Racket has been installed correctly.

```
#lang racket
(display "Hello, World!")
```

3- Racket Tutorial

3-1 Variables and Values

In Racket, variables are defined as follows:

```
#lang racket
(define x 10)
(define name "Ali")
(display name) ; It will display Ali
```

3-2 Operations

3-2-1 Basic Arithmetic Operators:

```
#lang racket
(+ 3 10)        ;add
(- 10 4)        ;sub
(* 6 8)         ;multiply
(/ 20 4)        ;division
(exp 2 3)       ;expt
(reminder 10 3) ;reminder
(quotient 10 3) ;quotient
```

- Any number of values can be used with a single operator:

```
#lang racket
(+ 1 2 3 4 5) ;result = 15
```

- In Racket, there is no need for additional parentheses to determine the order of operations because the syntax is prefix-based.

3-2-2 Comparison Operators:

```
#lang racket
(> 10 5)  ; #t
(< 3 8)   ; #t
(= 4 4)   ; #t
(>= 5 5)  ; #t
(<= 3 2)  ; #f
```

- The output of these functions is either #t (true) or #f (false).

3-2-3 Checking if a Number is Even or Odd:

```
#lang racket
(even? 4) ; #t
(odd? 4)  ; #f
```

3-2-4 Absolute Value:

```
#lang racket
(abs -5)  ; 5
(abs 10)  ; 10
```

3-2-5 Random Numbers:

```
#lang racket
(random 10)         ; [0,9]
(+ 5 (random 6))    ; [5,10]
```

3-2-6 Rounding Numbers:

```
#lang racket
(round 3.6)   ; 4
(round 3.4)   ; 3
(floor 3.9)   ; 3
(ceiling 3.1) ; 4
```

3-2-7 Advanced Math Functions (sqrt, sin, cos, log):

```
#lang racket
(sqrt 25) ; 5
(log 10)  ; 2.30258
(sin 0)   ; 0.0
(cos 0)   ; 1.0
```

3-3 Defining Functions

3-3-1 To define a function in Racket, we use **define** as follows:

```
(define (function_name parameter_1 parameter_2 ... parameter_n)
      function_body)
```

For example, to define a function that adds two numbers, we proceed as follows:

```
#lang racket
(define (sum a b)
      (+a b))
(display (sum 4 5)) ; it will show 9
```

3-3-2 You can define functions anonymously **lambda**:

```
(lambda (parameter_1 parameter_2 ... parameter_3)
      function_body)
```

For example, to define a function for multiplying two numbers, we can proceed as follows:

```
#lang racket
(define multiply (lambda (x y) (* x y)))
(display (multiply 4 5)) ; it will show 20
```

3-4 Arrays

3-4-1 Defining a List:

```
(define "list_name"(list "parameter_1" "parameter_2"..."parameter_3"))
```

For example:

```
#lang racket
(define my-list (list 1 2 3 4 5))
(display my-list) ; it  will show (1 2 3 4 5)
```

Or:

```
#lang racket
(define my-list ‘(1 2 3 4 5))
(display my-list) ; it  will show (1 2 3 4 5)
```

3-4-2 Adding an Element to a List:

cons allows us to add an element to the beginning of a list. Here, **cons** is used to add 1, 2, and 3 to the list.

```
#lang racket
(define my-list (cons 1 (cons 2 (cons 3 '()))))
(display my-list)  ; (1 2 3)
```

3-4-3 Accessing List Elements:

3-4-3-1 Using **first** to Access the First Element of a List

```
#lang racket
(define my-list '(1 2 3 4 5))
(display (first my-list)) ; it will show 1
```

3-4-3-2 Using **rest** to Access the List Except the First Element:

```
#lang racket
(define my-list '(1 2 3 4 5))
(display (rest my-list)) ; it will show (2 3 4 5)
```

3-4-3-3 Using **third** and **second** to Access Specific Elements:

```
#lang racket
(define my-list '(1 2 3 4 5))
```

```
(display (second my-list)) ; it will show 2
(display (third my-list))  ; it will show 3
```

3-4-3-4 Checking if a List is Empty:

```
#lang racket
(define my-list '(1 2 3 4 5))
(display (empty? my-list)) ; it will show #f
(display (empty ‘()))      ; it will show #t
```

3-4-3-5 Finding the Length of a List:

```
#lang racket
(define my-list '(1 2 3 4 5))
(display (length my-list)) ; it will show 5
```

3-4-4 Modifications to Lists:

3-4-4-1 Adding an Element to the Beginning of a List with **cons**:

```
#lang racket
(define my-list '(1 2 3 4 5))
(define new-list (cons 0 my-list))
(display new-list)  ; it will show (0 1 2 3 4 5)
```

3-4-4-2 Concatenating Two Lists:

```
#lang racket
(define list_1 '(1 2 3 4 5))
(define list_2 '(6 7 8 9 10))
(define new_list (append list_1 list_2))
(display new-list)  ; it will show (0 1 2 3 4 5)
```

3-4-5 Traversing Through Lists:

3-4-5-1 Using **map** to Apply a Function to All Elements of a List:

```
#lang racket
(define my-list (list 1 2 3 4 5))
(define squared-list (map (lambda (x) (* x x)) my-list))
(display squared-list) ; it  will show (1 4 9 16 25)
```

3-4-5-2 Using the **filter** Function to Filter Elements:

```
#lang racket
(define my-list (list 1 2 3 4 5))
(define even_number (filter even? my_list))
(display even_number) ; it  will show (2 4)
```

3-4-6 Converting a List to a String and Vice Versa:

3-4-6-1 Converting a List to a String:

```
#land racket
(define my-list '(H e l l o))
(display (list->string my-list))
```

3-4-6-2 Converting a String to a List:

```
#land racket
(define my-string "HELLO")
(display (string->list my-string))
```

3-5 Conditional Statements

3-5-1 The **cond** Statement:

If we need to check multiple different conditions, we use this statement, which has the following format:

```
(cond
      [condition_1 operation_1]
      [condition_2 operation_2]
      [condition_3 operation_3]
      .
      .
      .
      [else default operation])
```

For example:

```
#lang racket
```

```
(define x 5)
(cond
      [(> x 10) (display "x is greater than 10")]
      [(> x 3) (display "x is greater than 3")]
      [else (display "x is less or equal 3")])

;output is x is greater than 3
```

3-5-2 Using the **if** Statement:

A simpler conditional structure that is used to check a specific condition and execute one of two expressions based on the result of that condition is the **if** statement.

```
(if condition
Statement for when the condition is true
Statement for when the condition is false )
```

For example:

```
#lang racket
(define x 5)
(if (> x 3)
      (display "x is greater than 3")
      (display "x is less or equal than 3"))
```

3-5-3 Using **and** and **or**:

3-5-3-1 Using **and**:

```
#lang racket
(define x 5)
(define y 10)
(if (and (> x 3) (< y 20))
      (display "both condition are true")
      (display "at least one of the conditions is false"))
```

3-5-3-2 Using **or**:

```
#lang racket
(define x 5)
(define y 10)
(if (or (> x 3) (< y 20))
      (display "at least one of the conditions is true")
      (display "both condition are false"))
```

3-5-4 The **when** and **unless** Statements:

3-5-4-1 The **when** Statement:

The **when** statement is used when you want to perform a series of operations only when a condition is true. It only executes the operations if the condition evaluates to **#t**.

```
#lang racket
(define x 5)
(when (> x 3)
  (display "x is greater than 3"))
```

3-5-4-2 The **unless** Statement:

The **unless** statement is the opposite of **when**. It executes the expressions only when the condition is false. If the condition evaluates to **#f**, the operations inside **unless** are executed.

```
#lang racket
(define x 2)
(unless (> x 3)
  (display "x is less or equal than 3"))
```

3-6 Recursive Structure

- In Racket, recursion refers to calling a function within itself. This method is useful for solving problems that require repetition, especially when the number of iterations or their complexity is not known. To implement recursion, we typically have a **base case**, which prevents further function calls once it's reached, and a **recursive case**, where the function calls itself again.
- In Racket, loops are not commonly used, so recursion is often applied to solve problems such as traversing lists.

3-6-1 Factorial Example:

```
#lang racket
(define (factorial n)
      (if (= n 0)
            1
            (* n (factorial (- n 1)))))
(display (factorial 5))
```

3-6-2 Summing Numbers from 1 to n Example:

```
#lang racket
(define (sum n)
      (if (= n 0)
            0
            (+ n (sum (- n 1)))))
(display (sum 5))
```

3-6-3 Recursion with Lists (Summing All List Elements Example):

```
#lang racket
(define (sum-list input_list)
      (if (empty? input_list)
           0
           (+ (first input_list) (sum-list (rest input_list)))))
(display (sum-list `(1 2 3 4 5)))
```

3-7 Struct in Racket:

- In Racket, a **structure** (or **struct**) allows you to create more complex data types. These structures are similar to classes in object-oriented languages, but in Racket, they act more like custom data types that can have various fields. Using the **struct** keyword in Racket, you can define a new data type that includes fields and associated functions. These data types can act similarly to objects in other object-oriented programming languages.
  3-7-1 Displaying and Summing Elements of a Binary Tree Created with **struct** in Racket :

```
#lang racket

(struct node (value left right))

(define input_tree (node 3 (node 4 (node 7 (node 9 0 0) 0) 0) (node 5 0
0) ))
(define sum_result 0)

(define (illu tree)
 (cond
 ((node? tree)
 (display "(")
 (display (node-value tree))
 (display " ")
 (illu (node-left tree))
 (display " ")
 (illu (node-right tree))
 (display ")"))
 (else (display tree))))

(define (tree_sum_loop tree)
    (cond ((node? tree)
            (+ (+ (tree_sum_loop (node-left tree)) (tree_sum_loop
(node-right tree))) (node-value tree))
        )(else 0)
    )
)

(define (tree_sumation tree)
    (cond
        ((node? tree)
            (display (tree_sum_loop tree))
        )
        (else (display "input value must be tree"))
    )
)

(tree_sumation input_tree)
(display "\n")
(illu input_tree)
```

4- Sources:

4-1 For more examples, you can check out this repository on GitHub

4-2 Documentation on the official Racket website.

Thank you for your attention