Racket

Racket is a general purpose, multi-paradigm programming language in the Lisp/Scheme family.

Feedback is appreciated! You can reach me at [@th3rac25](http://twitter.com/th3rac25) or th3rac25 [at] [google's email service]

```
#lang racket ; defines the language we are using

;;; Comments

;; Single line comments start with a semicolon

#| Block comments
   can span multiple lines and...
    #|
       they can be nested!
    |#
|#

;; S-expression comments discard the following expression,
;; useful to comment expressions when debugging
#; (this expression is discarded)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 1. Primitive Datatypes and Operators
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Numbers
99999999999999999999 ; integers
#b111                ; binary => 7
#o111                ; octal => 73
#x111                ; hexadecimal => 273
3.14                 ; reals
6.02e+23
1/2                  ; rationals
1+2i                 ; complex numbers

;; Function application is written (f x y z ...)
;; where f is a function and x, y, z, ... are operands
;; If you want to create a literal list of data, use ' to stop it from
;; being evaluated
'(+ 1 2) ; => (+ 1 2)
;; Now, some arithmetic operations
(+ 1 1)  ; => 2
(- 8 1)  ; => 7
(* 10 2) ; => 20
(expt 2 3) ; => 8
(quotient 5 2) ; => 2
(remainder 5 2) ; => 1
(/ 35 5) ; => 7
(/ 1 3) ; => 1/3
(exact->inexact 1/3) ; => 0.3333333333333333
(+ 1+2i  2-3i) ; => 3-1i
```

```
;;; Booleans
#t ; for true
#f ; for false -- any value other than #f is true
(not #t) ; => #f
(and 0 #f (error "doesn't get here")) ; => #f
(or #f 0 (error "doesn't get here"))  ; => 0

;;; Characters
#\A ; => #\A
#\  ; => #\
#\u03BB ; => #\

;;; Strings are fixed-length array of characters.
"Hello, world!"
"Benjamin \"Bugsy\" Siegel"   ; backslash is an escaping character
"Foo\tbar\41\x21\u0021\a\r\n" ; includes C escapes, Unicode
" x:( .→).xx"              ; can include Unicode characters

;; Strings can be added too!
(string-append "Hello " "world!") ; => "Hello world!"

;; A string can be treated like a list of characters
(string-ref "Apple" 0) ; => #\A

;; format can be used to format strings:
(format "~a can be ~a" "strings" "formatted")

;; Printing is pretty easy
(printf "I'm Racket. Nice to meet you!\n")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 2. Variables
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; You can create a variable using define
;; a variable name can use any character except: ()[]{}",'`;#|\
(define some-var 5)
some-var ; => 5

;; You can also use unicode characters
(define  subset?)
(  (set 3 2) (set 1 2 3)) ; => #t

;; Accessing a previously unassigned variable is an exception
; x ; => x: undefined ...

;; Local binding: `me' is bound to "Bob" only within the (let ...)
(let ([me "Bob"])
  "Alice"
  me) ; => "Bob"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 3. Structs and Collections
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; Structs
(struct dog (name breed age))
(define my-pet
  (dog "lassie" "collie" 5))
my-pet ; => #<dog>
(dog? my-pet) ; => #t
(dog-name my-pet) ; => "lassie"

;;; Pairs (immutable)
;; `cons' constructs pairs, `car' and `cdr' extract the first
;; and second elements
(cons 1 2) ; => '(1 . 2)
(car (cons 1 2)) ; => 1
(cdr (cons 1 2)) ; => 2

;;; Lists

;; Lists are linked-list data structures, made of `cons' pairs and end
;; with a `null' (or '()) to mark the end of the list
(cons 1 (cons 2 (cons 3 null))) ; => '(1 2 3)
;; `list' is a convenience variadic constructor for lists
(list 1 2 3) ; => '(1 2 3)
;; and a quote can also be used for a literal list value
'(1 2 3) ; => '(1 2 3)

;; Can still use `cons' to add an item to the beginning of a list
(cons 4 '(1 2 3)) ; => '(4 1 2 3)

;; Use `append' to add lists together
(append '(1 2) '(3 4)) ; => '(1 2 3 4)

;; Lists are a very basic type, so there is a *lot* of functionality for
;; them, a few examples:
(map add1 '(1 2 3))          ; => '(2 3 4)
(map + '(1 2 3) '(10 20 30)) ; => '(11 22 33)
(filter even? '(1 2 3 4))    ; => '(2 4)
(count even? '(1 2 3 4))     ; => 2
(take '(1 2 3 4) 2)          ; => '(1 2)
(drop '(1 2 3 4) 2)          ; => '(3 4)

;;; Vectors

;; Vectors are fixed-length arrays
#(1 2 3) ; => '#(1 2 3)

;; Use `vector-append' to add vectors together
(vector-append #(1 2 3) #(4 5 6)) ; => #(1 2 3 4 5 6)

;;; Sets

;; Create a set from a list
(list->set '(1 2 3 1 2 3 3 2 1 3 2 1)) ; => (set 1 2 3)

;; Add a member with `set-add'
```

```
;; (Functional: returns the extended set rather than mutate the input)
(set-add (set 1 2 3) 4) ; => (set 1 2 3 4)

;; Remove one with `set-remove'
(set-remove (set 1 2 3) 1) ; => (set 2 3)

;; Test for existence with `set-member?'
(set-member? (set 1 2 3) 1) ; => #t
(set-member? (set 1 2 3) 4) ; => #f

;;; Hashes

;; Create an immutable hash table (mutable example below)
(define m (hash 'a 1 'b 2 'c 3))

;; Retrieve a value
(hash-ref m 'a) ; => 1

;; Retrieving a non-present value is an exception
; (hash-ref m 'd) => no value found

;; You can provide a default value for missing keys
(hash-ref m 'd 0) ; => 0

;; Use `hash-set' to extend an immutable hash table
;; (Returns the extended hash instead of mutating it)
(define m2 (hash-set m 'd 4))
m2 ; => '#hash((b . 2) (a . 1) (d . 4) (c . 3))

;; Remember, these hashes are immutable!
m ; => '#hash((b . 2) (a . 1) (c . 3))  <-- no `d'

;; Use `hash-remove' to remove keys (functional too)
(hash-remove m 'a) ; => '#hash((b . 2) (c . 3))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 3. Functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Use `lambda' to create functions.
;; A function always returns the value of its last expression
(lambda () "Hello World") ; => #<procedure>
;; Can also use a unicode ` '
( () "Hello World")     ; => same function

;; Use parens to call all functions, including a lambda expression
((lambda () "Hello World")) ; => "Hello World"
((  () "Hello World"))      ; => "Hello World"

;; Assign a function to a var
(define hello-world (lambda () "Hello World"))
(hello-world) ; => "Hello World"

;; You can shorten this using the function definition syntactic sugar:
```

```
(define (hello-world2) "Hello World")

;; The () in the above is the list of arguments for the function
(define hello
  (lambda (name)
    (string-append "Hello " name)))
(hello "Steve") ; => "Hello Steve"
;; ... or equivalently, using a sugared definition:
(define (hello2 name)
  (string-append "Hello " name))

;; You can have multi-variadic functions too, using `case-lambda'
(define hello3
  (case-lambda
    [() "Hello World"]
    [(name) (string-append "Hello " name)]))
(hello3 "Jake") ; => "Hello Jake"
(hello3) ; => "Hello World"
;; ... or specify optional arguments with a default value expression
(define (hello4 [name "World"])
  (string-append "Hello " name))

;; Functions can pack extra arguments up in a list
(define (count-args . args)
  (format "You passed ~a args: ~a" (length args) args))
(count-args 1 2 3) ; => "You passed 3 args: (1 2 3)"
;; ... or with the unsugared `lambda' form:
(define count-args2
  (lambda args
    (format "You passed ~a args: ~a" (length args) args)))

;; You can mix regular and packed arguments
(define (hello-count name . args)
  (format "Hello ~a, you passed ~a extra args" name (length args)))
(hello-count "Finn" 1 2 3)
; => "Hello Finn, you passed 3 extra args"
;; ... unsugared:
(define hello-count2
  (lambda (name . args)
    (format "Hello ~a, you passed ~a extra args" name (length args))))

;; And with keywords
(define (hello-k #:name [name "World"] #:greeting [g "Hello"] . args)
  (format "~a ~a, ~a extra args" g name (length args)))
(hello-k)                  ; => "Hello World, 0 extra args"
(hello-k 1 2 3)            ; => "Hello World, 3 extra args"
(hello-k #:greeting "Hi") ; => "Hi World, 0 extra args"
(hello-k #:name "Finn" #:greeting "Hey") ; => "Hey Finn, 0 extra args"
(hello-k 1 2 3 #:greeting "Hi" #:name "Finn" 4 5 6)
                                         ; => "Hi Finn, 6 extra args"

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 4. Equality
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; for numbers use `='
(= 3 3.0) ; => #t
(= 2 1)   ; => #f

;; `eq?' returns #t if 2 arguments refer to the same object (in memory),
;; #f otherwise.
;; In other words, it's a simple pointer comparison.
(eq? '() '()) ; => #t, since there exists only one empty list in memory
(let ([x '()] [y '()])
  (eq? x y))  ; => #t, same as above

(eq? (list 3) (list 3)) ; => #f
(let ([x (list 3)] [y (list 3)])
  (eq? x y))            ; => #f - not the same list in memory!

(let* ([x (list 3)] [y x])
  (eq? x y)) ; => #t, since x and y now point to the same stuff

(eq? 'yes 'yes) ; => #t
(eq? 'yes 'no)  ; => #f

(eq? 3 3)   ; => #t - be careful here
            ; It's better to use `=' for number comparisons.
(eq? 3 3.0) ; => #f

(eq? (expt 2 100) (expt 2 100))               ; => #f
(eq? (integer->char 955) (integer->char 955)) ; => #f

(eq? (string-append "foo" "bar") (string-append "foo" "bar")) ; => #f

;; `eqv?' supports the comparison of number and character datatypes.
;; for other datatypes, `eqv?' and `eq?' return the same result.
(eqv? 3 3.0)                                  ; => #f
(eqv? (expt 2 100) (expt 2 100))              ; => #t
(eqv? (integer->char 955) (integer->char 955)) ; => #t

(eqv? (string-append "foo" "bar") (string-append "foo" "bar"))   ; => #f

;; `equal?' supports the comparison of the following datatypes:
;; strings, byte strings, pairs, mutable pairs, vectors, boxes,
;; hash tables, and inspectable structures.
;; for other datatypes, `equal?' and `eqv?' return the same result.
(equal? 3 3.0)                                                  ; => #f
(equal? (string-append "foo" "bar") (string-append "foo" "bar")) ; => #t
(equal? (list 3) (list 3))                                      ; => #t

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 5. Control Flow
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Conditionals

(if #t               ; test expression
```

```
    "this is true"   ; then expression
    "this is false") ; else expression
; => "this is true"

;; In conditionals, all non-#f values are treated as true
(member 'Groucho '(Harpo Groucho Zeppo)) ; => '(Groucho Zeppo)
(if (member 'Groucho '(Harpo Groucho Zeppo))
    'yep
    'nope)
; => 'yep

;; `cond' chains a series of tests to select a result
(cond [(> 2 2) (error "wrong!")]
      [(< 2 2) (error "wrong again!")]
      [else 'ok]) ; => 'ok

;;; Pattern Matching

(define (fizzbuzz? n)
  (match (list (remainder n 3) (remainder n 5))
    [(list 0 0) 'fizzbuzz]
    [(list 0 _) 'fizz]
    [(list _ 0) 'buzz]
    [_          #f]))

(fizzbuzz? 15) ; => 'fizzbuzz
(fizzbuzz? 37) ; => #f

;;; Loops

;; Looping can be done through (tail-) recursion
(define (loop i)
  (when (< i 10)
    (printf "i=~a\n" i)
    (loop (add1 i))))
(loop 5) ; => i=5, i=6, ...

;; Similarly, with a named let
(let loop ((i 0))
  (when (< i 10)
    (printf "i=~a\n" i)
    (loop (add1 i)))) ; => i=0, i=1, ...

;; See below how to add a new `loop' form, but Racket already has a very
;; flexible `for' form for loops:
(for ([i 10])
  (printf "i=~a\n" i)) ; => i=0, i=1, ...
(for ([i (in-range 5 10)])
  (printf "i=~a\n" i)) ; => i=5, i=6, ...

;;; Iteration Over Other Sequences
;; `for' allows iteration over many other kinds of sequences:
;; lists, vectors, strings, sets, hash tables, etc...
```

```
(for ([i (in-list '(l i s t))])
  (displayln i))

(for ([i (in-vector #(v e c t o r))])
  (displayln i))

(for ([i (in-string "string")])
  (displayln i))

(for ([i (in-set (set 'x 'y 'z))])
  (displayln i))

(for ([(k v) (in-hash (hash 'a 1 'b 2 'c 3 ))])
  (printf "key:~a value:~a\n" k v))

;;; More Complex Iterations

;; Parallel scan of multiple sequences (stops on shortest)
(for ([i 10] [j '(x y z)]) (printf "~a:~a\n" i j))
; => 0:x 1:y 2:z

;; Nested loops
(for* ([i 2] [j '(x y z)]) (printf "~a:~a\n" i j))
; => 0:x, 0:y, 0:z, 1:x, 1:y, 1:z

;; Conditions
(for ([i 1000]
      #:when (> i 5)
      #:unless (odd? i)
      #:break (> i 10))
  (printf "i=~a\n" i))
; => i=6, i=8, i=10

;;; Comprehensions
;; Very similar to `for' loops -- just collect the results

(for/list ([i '(1 2 3)])
  (add1 i)) ; => '(2 3 4)

(for/list ([i '(1 2 3)] #:when (even? i))
  i) ; => '(2)

(for/list ([i 10] [j '(x y z)])
  (list i j)) ; => '((0 x) (1 y) (2 z))

(for/list ([i 1000] #:when (> i 5) #:unless (odd? i) #:break (> i 10))
  i) ; => '(6 8 10)

(for/hash ([i '(1 2 3)])
  (values i (number->string i)))
; => '#hash((1 . "1") (2 . "2") (3 . "3"))

;; There are many kinds of other built-in ways to collect loop values:
(for/sum ([i 10]) (* i i)) ; => 285
```

```
(for/product ([i (in-range 1 11)]) (* i i)) ; => 13168189440000
(for/and ([i 10] [j (in-range 10 20)]) (< i j)) ; => #t
(for/or ([i 10] [j (in-range 0 20 2)]) (= i j)) ; => #t
;; And to use any arbitrary combination, use `for/fold'
(for/fold ([sum 0]) ([i '(1 2 3 4)]) (+ sum i)) ; => 10
;; (This can often replace common imperative loops)

;;; Exceptions

;; To catch exceptions, use the `with-handlers' form
(with-handlers ([exn:fail? (lambda (exn) 999)])
  (+ 1 "2")) ; => 999
(with-handlers ([exn:break? (lambda (exn) "no time")])
  (sleep 3)
  "phew") ; => "phew", but if you break it => "no time"

;; Use `raise' to throw exceptions or any other value
(with-handlers ([number?    ; catch numeric values raised
                 identity]) ; return them as plain values
  (+ 1 (raise 2))) ; => 2

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 6. Mutation
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Use `set!' to assign a new value to an existing variable
(define n 5)
(set! n (add1 n))
n ; => 6

;; Use boxes for explicitly mutable values (similar to pointers or
;; references in other languages)
(define n* (box 5))
(set-box! n* (add1 (unbox n*)))
(unbox n*) ; => 6

;; Many Racket datatypes are immutable (pairs, lists, etc), some come in
;; both mutable and immutable flavors (strings, vectors, hash tables,
;; etc...)

;; Use `vector' or `make-vector' to create mutable vectors
(define vec (vector 2 2 3 4))
(define wall (make-vector 100 'bottle-of-beer))
;; Use vector-set! to update a slot
(vector-set! vec 0 1)
(vector-set! wall 99 'down)
vec ; => #(1 2 3 4)

;; Create an empty mutable hash table and manipulate it
(define m3 (make-hash))
(hash-set! m3 'a 1)
(hash-set! m3 'b 2)
(hash-set! m3 'c 3)
(hash-ref m3 'a)   ; => 1
```

```
(hash-ref m3 'd 0) ; => 0
(hash-remove! m3 'a)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 7. Modules
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Modules let you organize code into multiple files and reusable
;; libraries; here we use sub-modules, nested in the whole module that
;; this text makes (starting from the "#lang" line)

(module cake racket/base ; define a `cake' module based on racket/base

  (provide print-cake) ; function exported by the module

  (define (print-cake n)
    (show "   ~a   " n #\.)
    (show " .-~a-. " n #\|)
    (show " | ~a | " n #\space)
    (show "----a---" n #\-))

  (define (show fmt n ch) ; internal function
    (printf fmt (make-string n ch))
    (newline)))

;; Use `require' to get all `provide'd names from a module
(require 'cake) ; the ' is for a local submodule
(print-cake 3)
; (show "~a" 1 #\A) ; => error, `show' was not exported

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 8. Classes and Objects
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Create a class fish% (-% is idiomatic for class bindings)
(define fish%
  (class object%
    (init size) ; initialization argument
    (super-new) ; superclass initialization
    ;; Field
    (define current-size size)
    ;; Public methods
    (define/public (get-size)
      current-size)
    (define/public (grow amt)
      (set! current-size (+ amt current-size)))
    (define/public (eat other-fish)
      (grow (send other-fish get-size)))))

;; Create an instance of fish%
(define charlie
  (new fish% [size 10]))

;; Use `send' to call an object's methods
```

```
(send charlie get-size) ; => 10
(send charlie grow 6)
(send charlie get-size) ; => 16

;; `fish%' is a plain "first class" value, which can get us mixins
(define (add-color c%)
  (class c%
    (init color)
    (super-new)
    (define my-color color)
    (define/public (get-color) my-color)))
(define colored-fish% (add-color fish%))
(define charlie2 (new colored-fish% [size 10] [color 'red]))
(send charlie2 get-color)
;; or, with no names:
(send (new (add-color fish%) [size 10] [color 'red]) get-color)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 9. Macros
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Macros let you extend the syntax of the language

;; Let's add a while loop
(define-syntax-rule (while condition body ...)
  (let loop ()
    (when condition
      body ...
      (loop))))

(let ([i 0])
  (while (< i  10)
    (displayln i)
    (set! i (add1 i))))

;; Macros are hygienic, you cannot clobber existing variables!
(define-syntax-rule (swap! x y) ; -! is idiomatic for mutation
  (let ([tmp x])
    (set! x y)
    (set! y tmp)))

(define tmp 2)
(define other 3)
(swap! tmp other)
(printf "tmp = ~a; other = ~a\n" tmp other)
;; The variable `tmp` is renamed to `tmp_1`
;; in order to avoid name conflict
;; (let ([tmp_1 tmp])
;;   (set! tmp other)
;;   (set! other tmp_1))

;; But they are still code transformations, for example:
(define-syntax-rule (bad-while condition body ...)
  (when condition
```

```
    body ...
    (bad-while condition body ...)))
;; this macro is broken: it generates infinite code, if you try to use
;; it, the compiler will get in an infinite loop

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 10. Contracts
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Contracts impose constraints on values exported from modules

(module bank-account racket
  (provide (contract-out
            [deposit (-> positive? any)] ; amounts are always positive
            [balance (-> positive?)]))

  (define amount 0)
  (define (deposit a) (set! amount (+ amount a)))
  (define (balance) amount)
  )

(require 'bank-account)
(deposit 5)

(balance) ; => 5

;; Clients that attempt to deposit a non-positive amount are blamed
;; (deposit -5) ; => deposit: contract violation
;; expected: positive?
;; given: -5
;; more details....

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 11. Input & output
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Racket has this concept of "port", which is very similar to file
;; descriptors in other languages

;; Open "/tmp/tmp.txt" and write "Hello World"
;; This would trigger an error if the file's already existed
(define out-port (open-output-file "/tmp/tmp.txt"))
(displayln "Hello World" out-port)
(close-output-port out-port)

;; Append to "/tmp/tmp.txt"
(define out-port (open-output-file "/tmp/tmp.txt"
                                   #:exists 'append))
(displayln "Hola mundo" out-port)
(close-output-port out-port)

;; Read from the file again
(define in-port (open-input-file "/tmp/tmp.txt"))
(displayln (read-line in-port))
```

```
; => "Hello World"
(displayln (read-line in-port))
; => "Hola mundo"
(close-input-port in-port)

;; Alternatively, with call-with-output-file you don't need to explicitly
;; close the file
(call-with-output-file "/tmp/tmp.txt"
  #:exists 'update ; Rewrite the content
  (  (out-port)
    (displayln "World Hello!" out-port)))

;; And call-with-input-file does the same thing for input
(call-with-input-file "/tmp/tmp.txt"
  (  (in-port)
    (displayln (read-line in-port))))
```

Further Reading

Still up for more? Try Getting Started with Racket