Racket Programming Cheatsheet

A concise reference for Racket programming, covering syntax, data structures, control flow, and module system basics.

Racket Basics & Syntax

Core Syntax

Expressions	Racket code consists of expressions. Expressions can be literals, identifiers, or procedure applications.
Procedure Application	`(procedure-name arg1 arg2 ...)` Applies a procedure to arguments. Parentheses are *required*.
Defining Variables	`(define variable-name expression)` Binds a value to a variable.
Comments	`;` - Single-line comment. `#| ... |#` - Multi-line comment.
Boolean Values	`#t` (true) and `#f` (false). Everything except `#f` is considered true in conditional contexts.
Number Types	Racket supports integers, rational numbers, real numbers, and complex numbers.

Basic Data Types

Numbers: Integers, decimals, fractions.

Booleans: `#t` (true), `#f` (false).

Strings: Enclosed in double quotes, e.g., `"hello"`.

Symbols: Quoted identifiers, e.g., `'symbol`.

Lists: Ordered collections, e.g., `'(1 2 3)`.

Vectors: Fixed-size arrays, e.g., `#(1 2 3)`.

Common Procedures

`(+ x y ...)`	Addition.
`(- x y ...)`	Subtraction.
`(* x y ...)`	Multiplication.
`(/ x y ...)`	Division.
`(= x y ...)`	Numerical equality.
`(< x y ...)`	Less than.

Control Flow

Conditionals

`(if condition then-expression else-expression)`
Evaluates `condition`. If true (not `#f`), evaluates `then-expression`; otherwise, evaluates `else-expression`.

`(cond [condition expression] ... [else expression])`
A series of conditional clauses. Evaluates the `condition` of each clause until one is true, then evaluates the corresponding `expression`. The `else` clause is optional and provides a default.

Boolean Logic

`(and expr ...)`	Returns `#t` if all expressions are true (not `#f`), otherwise `#f`. Short-circuits.
`(or expr ...)`	Returns `#t` if at least one expression is true (not `#f`), otherwise `#f`. Short-circuits.
`(not expr)`	Returns `#t` if `expr` is `#f`, otherwise `#f`.

Iteration

`(for ([item sequence]) body ...)`
Iterates over a `sequence` (e.g., a list or vector), binding each element to `item` and evaluating `body` in each iteration.

`(for/list ([item sequence]) body ...)`
Similar to `for`, but collects the results of evaluating `body` into a list.

(let loop ([var1 init1] [var2 init2] ...) body)
Defines a local recursive loop.

Data Structures

Lists

`(list arg ...)`	Creates a new list containing the given arguments.
`(cons first rest)`	Constructs a new list by adding `first` to the beginning of `rest` (which must be a list).
`(car list)`	Returns the first element of `list`.
`(cdr list)`	Returns the rest of `list` (excluding the first element).
`(null? list)`	Returns `#t` if `list` is empty, otherwise `#f`.
`(length list)`	Returns the number of elements in `list`.

Vectors

`(vector arg ...)`	Creates a new vector containing the given arguments.
`(vector-ref vector index)`	Returns the element at `index` in `vector`.
`(vector-set! vector index value)`	Sets the element at `index` in `vector` to `value`. Vectors are mutable.
`(vector-length vector)`	Returns the number of elements in `vector`.
`(vector? obj)`	Returns `#t` if `obj` is a vector, otherwise `#f`.

Hash Tables

`(make-hash)`	Creates a new empty hash table.
`(hash-set! hash key value)`	Associates `key` with `value` in `hash`.
`(hash-ref hash key)`	Returns the value associated with `key` in `hash`. Raises an error if the key is not found.
`(hash-ref hash key default-value)`	Returns the value associated with `key` in `hash`. Returns `default-value` if the key is not found.
`(hash-remove! hash key)`	Removes the association for `key` in `hash`.

Modules

Module Definition

`#lang racket`
Specifies the language (in this case, Racket) for the module. This should be the first line of your file.

`(module name language ... body ...)`
Defines a module named `name` using the specified `language`. The `body` contains definitions, expressions, and imports.

Exports

`(provide identifier ...)`
Specifies which identifiers (variables, procedures, etc.) are exported from the module and made available to other modules.

`(provide (all-defined-out))`
Exports all identifiers defined within the module.

Imports

`(require module-path ...)`
Imports identifiers from the specified `module-path`. `module-path` can be a file path, a library name, or a module name.

`(require (prefix id module-path))`
Imports identifiers from `module-path`, prefixing each with `id`.

`(require (only-in module-path id ...))`
Imports only the specified identifiers from `module-path`.