Functional Programming

Expressions, functions and cases in Racket

Last modified: 2026-11-13 13:37:25 (PST)

> **Source**
>
> Kameron Decker Harris's Fall 2026 syllabus opens the course with an introduction to functional programming; this chapter, its examples and its exercises are new.
> The definition of an expression is on the expressions and algorithms page, and those of a function, an argument and a value are on the functions page. The other data science notes link to both pages.

Expressions

Remark 1 (Running Racket). This course writes its programs in Racket, a programming language that grew out of Scheme, a dialect of Lisp. Racket comes with DrRacket, the program you use to write and run Racket code, and Lab 1 walks through using it. Each Racket file in this course starts with the line `#lang racket`. The Racket Guide is the reference for anything these notes do not cover.

Exercise 1 (Operation first)**.** Racket writes $7 - 2$ as `(- 7 2)`, with the operation first and the parentheses around the whole expression. Guess what each Racket expression returns:

1. `(* (+ 1 2) (- 10 4))`
2. `(- 20 (* 3 (+ 1 1)))`
3. `(+ 1 2 3 4)`

Then write $(2 + 3) \cdot 4 - 1$ in Racket.

Solution

Solution 1. Work from the innermost parentheses outward.

1. `(+ 1 2)` is 3 and `(- 10 4)` is 6, so the expression is `(* 3 6)`, which returns 18.
2. `(+ 1 1)` is 2, so `(* 3 (+ 1 1))` is 6, and `(- 20 6)` returns 14.
3. `+` can take any number of operands, and adds them all, so the expression returns 10.

The expression $(2 + 3) \cdot 4 - 1$ subtracts 1 from the product of $2 + 3$ and 4, so in Racket it is `(- (* (+ 2 3) 4) 1)`, which returns 19.
Racket returns the same values:

```
(* (+ 1 2) (- 10 4))
; 18
(- 20 (* 3 (+ 1 1)))
; 14
(+ 1 2 3 4)
; 10
(- (* (+ 2 3) 4) 1)
; 19
```

Definition 1 (Function, map, mapping, transformation)**.** A **function** f from a set A to a set B, (Wikipedia contributors 2026b) written $f : A \to B$, assigns to each element $a \in A$ exactly one element $f(a) \in B$.
A function is also called a **map** (or **mapping**) or a **transformation**; on this site the three words mean the same thing, although some authors reserve "transformation" for a function from a set to itself.

Definition 2 (Expression)**.** An **expression** (Wikipedia contributors 2026a) is a string of symbols, built from constants, variables, operations and function names according to the rules of the notation.

Definition 3 (Denotes)**.** An expression **denotes** a function when evaluating the expression at each input of the function, with the expression's variables taking that input, gives the function's value at that input.

Definition 4 (Racket expression, prefix notation)**.** A **Racket expression** (Definition 2) is one of:

- a constant, such as the number `7`;
- a name, such as `x` or `+`;
- an **application** `(f e1 ... en)`: an expression `f` for an operation or function, followed by its operands `e1`, …, `en`, which are expressions too, all inside one pair of parentheses.

Writing the operation before its operands is called prefix notation. Racket needs no precedence rules, because the nesting of the parentheses says which operation comes first: `(* (+ 2 3) 4)` adds first, and `(+ 2 (* 3 4))` multiplies first.

Remark 2 (How Racket evaluates an application). To **evaluate** an application `(f e1 ... en)` (Definition 4), Racket evaluates each operand `e1`, …, `en` to a value, then applies `f` to those values. In part 1 of Exercise 1:

```
(* (+ 1 2) (- 10 4))
= (* 3 (- 10 4))
= (* 3 6)
= 18
```

Each `=` replaces one application by its value, and the evaluation ends with a value, which cannot be simplified further.

Functions

Exercise 2 (Reading a definition)**.** This Racket code defines a function named `h`:

```
(define (h x)
  (- (* x x) (* 2 x)))
```

1. Write $h(x)$ as a formula in ordinary mathematical notation.
2. What does `(h 5)` return?
3. What does `(h (h 3))` return?

Solution

Solution 2.

1. $h(x) = x^2 - 2x$, since `(* x x)` is x^2 and `(* 2 x)` is $2x$.
2. $h(5) = 25 - 10 = 15$, so `(h 5)` returns 15.
3. Racket first evaluates the argument `(h 3)`, which is $9 - 6 = 3$, and then applies `h` to it, so `(h (h 3))` is `(h 3)` again and returns 3.

Racket returns the same values:

```
(h 5)
; 15
(h (h 3))
; 3
```

Definition 5 (Argument, input)**.** An **argument** (or **input**) of a function $f : A \to B$ is an element $a \in A$ that f is applied to.

Definition 6 (Value, output)**.** The **value** (or **output**) of a function $f : A \to B$ at an argument $a \in A$ is the element $f(a) \in B$ that f assigns to a.

Definition 7 (Defining a function in Racket)**.** The Racket definition

```
(define (f x1 ... xn)
  body)
```

defines a function named `f` with **parameters** `x1`, …, `xn`: names that stand for its arguments (Definition 5). To evaluate an application `(f a1 ... an)`, Racket evaluates the arguments `a1`, …, `an`, replaces each parameter in `body` by the value of its argument, and evaluates the result. The value of the application is the value (Definition 6) of `f` at those arguments.

Example 1 (Replacing a parameter by its argument)**.** With `h` from Exercise 2, Definition 7 evaluates `(h 5)` in four steps:

```
(h 5)
= (- (* 5 5) (* 2 5))
= (- 25 (* 2 5))
= (- 25 10)
= 15
```

The first step replaces `x` by 5 in the body of `h`, just as computing $h(5)$ replaces x by 5 in $x^2 - 2x$. Each later step replaces one application by its value (Remark 2).

Cases

Exercise 3 (Choosing a case)**.** A museum sets its ticket price, in dollars, by age:

```
(define (ticket-price age)
  (cond [(< age 5) 0]
        [(< age 18) 8]
        [else 12]))
```

Each line in square brackets pairs a test with a value.

1. Guess what `(ticket-price 4)`, `(ticket-price 5)`, `(ticket-price 17)` and `(ticket-price 70)` return.

2. Write the function as a formula with cases, for a whole number of years $a \geq 0$.

Solution

Solution 3.

1. Racket tries the tests in order and returns the value beside the first true one:
   - `(ticket-price 4)` returns 0, since $4 < 5$.
   - `(ticket-price 5)` returns 8, since $5 < 5$ is false but $5 < 18$ is true.
   - `(ticket-price 17)` returns 8, since $17 < 18$.
   - `(ticket-price 70)` returns 12, since neither test is true, so the `else` line applies.
2. The second case holds only when the first test is false, so its condition is $5 \leq a < 18$, not just $a < 18$:

$$p(a) = \begin{cases} 0 & \text{if } a < 5, \\ 8 & \text{if } 5 \leq a < 18, \\ 12 & \text{if } a \geq 18. \end{cases}$$

Racket returns the same values:

```
(ticket-price 4)
; 0
(ticket-price 5)
; 8
(ticket-price 17)
; 8
(ticket-price 70)
; 12
```

Definition 8 (True and false in Racket)**.** Racket writes the two truth values as `#t` (true) and `#f` (false). A comparison such as `(< 4 5)` or `(= 3 7)` returns one of them, and `and`, `or` and `not` combine them: `(and (< 1 2) (< 3 2))` returns `#f`.

Definition 9 (`if` and `cond`)**.** The expression `(if test a b)` evaluates `test`, and returns the value of `a` if `test` is true and the value of `b` if it is false.
The expression

```
(cond [test1 a1]
      [test2 a2]
      ...
      [else an])
```

evaluates the tests in order and returns the value of the expression beside the first true one, or the value of `an` if none is true. Like a function defined by cases in mathematics, each case of a `cond` applies only when every case before it fails.
`if` and `cond` look like applications (Definition 4), but they are not: an application evaluates all of its operands (Remark 2), while `if` and `cond` evaluate only the tests they need and the one branch they return. `and`, `or` and `define` are not applications either.

Example 2 (Absolute value with `if`)**.** The absolute value

$$|x| = \begin{cases} -x & \text{if } x < 0, \\ x & \text{otherwise} \end{cases}$$

has two cases, so one `if` (Definition 9) is enough:

```
(define (absolute x)
  (if (< x 0)
      (- x)
      x))
```

Applied to -7 and to 2, it returns their absolute values:

```
(absolute -7)
; 7
(absolute 2)
; 2
```

Racket already has this function, under the name `abs`.

Pure functions

Exercise 4 (Same call, different answer)**.** This Python code hands out ID numbers:

```
count = 0

def next_id():
    global count
    count = count + 1
    return count
```

1. The program calls `next_id()` twice in a row. What does each call return?
2. Is there a mathematical function f (Definition 1) with no arguments whose value is what `next_id()` returns every time it is called?

Solution

Solution 4.

1. The first call changes `count` from 0 to 1 and returns 1. The second call changes `count` from 1 to 2 and returns 2.
2. No. A function with no arguments has exactly one value, but `next_id()` returned 1 and then 2. What it returns depends on the variable `count`, which is not an argument and which each call changes.

Definition 10 (Side effect)**.** A **side effect** of a piece of code is any change it makes besides returning a value, such as changing a variable, printing, or writing a file. `next_id` in Exercise 4 has a side effect: it changes `count`.

Definition 11 (Pure function)**.** A function in a program is **pure** (Wikipedia contributors 2026e) if

- it returns the same value whenever it is called with the same arguments, and
- it has no side effects (Definition 10).

A pure function computes a mathematical function (Definition 1) of its arguments.

Definition 12 (Functional programming)**. Functional programming** (Wikipedia contributors 2026c) is a style of programming in which a program is built by defining and applying pure functions (Definition 11), rather than by running statements that change variables.

Remark 3 (Why this course uses functional programming)*.* In a program built from pure functions (Definition 11), an expression can be replaced by its value anywhere it appears without changing what the program does. That is what makes evaluation by substitution (Example 1) work, and it is also how algebra works: $h(5)$ can be replaced by 15 in any formula. So a pure function can be checked with the same reasoning as a formula, and this course proves facts about programs that way.
`next_id` in Exercise 4 is not pure, so `next_id() + next_id()` cannot be simplified to `2 * next_id()`. On a fresh start, `next_id() + next_id()` returns 3, from $1 + 2$, but `2 * next_id()` returns 2. Racket can change variables too, with `set!`, but this course does not use it.

Functions as values

Exercise 5 (A function as an argument)**.** The function `twice` takes a function `f` and a value `x`:

```
(define (twice f x)
  (f (f x)))

(define (add-three x)
  (+ x 3))
```

Guess what each expression returns:

1. `(twice add-three 10)`
2. `(twice h 4)`, with `h` from Exercise 2
3. `(twice add-three (twice add-three 0))`

Solution

Solution 5. Replace `f` and `x` in the body `(f (f x))` (Definition 7):

1. `(twice add-three 10)` is `(add-three (add-three 10))`, which is `(add-three 13)`, which returns 16.
2. `(twice h 4)` is `(h (h 4))`. Since $h(4) = 16 - 8 = 8$ and $h(8) = 64 - 16 = 48$, it returns 48.
3. The argument `(twice add-three 0)` returns 6, so the whole expression is `(twice add-three 6)`, which returns 12.

Racket returns the same values:

```
(twice add-three 10)
; 16
(twice h 4)
; 48
(twice add-three (twice add-three 0))
; 12
```

Definition 13 (Higher-order function)**.** A **higher-order function** (Wikipedia contributors 2026d) is a function that takes a function as an argument, or returns a function as its value. `twice` in Exercise 5 is a higher-order function.

Example 3 (`map` and `filter`)**.** Racket's `(list 5 7 9)` makes the list of 5, 7 and 9, which Racket prints with a quote mark in front:

```
(list 5 7 9)
; '(5 7 9)
```

Two built-in higher-order functions (Definition 13) work on lists:

- `(map f L)` applies `f` to each element of the list `L`, and returns the list of results.
- `(filter p L)` returns the list of the elements `x` of `L` for which `(p x)` is true.

Applied to `add-three` from Exercise 5 and to Racket's test `even?`, they return:

```
(map add-three (list 5 7 9))
; '(8 10 12)
(filter even? (list 5 6 7 8))
; '(6 8)
```

Neither one changes `L`; each returns a new list.

Exercise 6 (Applying a function to a list)**.** `(apply f L)` applies the function `f` to the elements of the list `L` as its arguments, so `(apply + (list 2 7 6))` is the same as `(+ 2 7 6)`:

```
(apply + (list 2 7 6))
; 15
(+ 2 7 6)
; 15
```

1. Guess what `(apply max (list 3 9 2))` returns.
2. Use `apply`, `max` and `min` to define `(spread L)`, which returns the largest element of a nonempty list of numbers minus its smallest element.
3. Guess what `(spread (list 4 -1 7 2))` returns.

Solution

Solution 6.

1. `(apply max (list 3 9 2))` is `(max 3 9 2)`, which returns 9.
2. One answer:

   ```
   (define (spread L)
     (- (apply max L) (apply min L)))
   ```

3. The largest element is 7 and the smallest is -1, so `(spread (list 4 -1 7 2))` returns $7 - (-1) = 8$.

`spread` needs no recursion: `max` and `min` already take any number of arguments, and `apply` hands each of them the whole list.
Racket returns the same values:

```
(apply max (list 3 9 2))
; 9
(spread (list 4 -1 7 2))
; 8
```

Anonymous functions and local names

Exercise 7 (A function with no name)**.** The function `pair-with` takes a list `L` and a value `y`:

```
(define (pair-with L y)
  (map (lambda (x) (list x y)) L))
```

Guess what each expression returns:

1. `((lambda (x) (* x x)) 7)`
2. `(map (lambda (x) (* 2 x)) (list 1 2 3))`
3. `(pair-with (list 1 2 3) "a")`

Solution

Solution 7. `(lambda (x) (* x x))` is the function that squares its argument, written in place without a name.

1. Applying it to 7 returns 49.
2. `map` (Example 3) applies the doubling function to each element, and returns `'(2 4 6)`.
3. In the body of `pair-with`, `y` is `"a"`, so `map` applies the function $x \mapsto (x, \text{"a"})$ to each element, and returns `'((1 "a") (2 "a") (3 "a"))`.

Racket returns the same values:

```
((lambda (x) (* x x)) 7)
; 49
(map (lambda (x) (* 2 x)) (list 1 2 3))
; '(2 4 6)
(pair-with (list 1 2 3) "a")
; '((1 "a") (2 "a") (3 "a"))
```

Definition 14 (`lambda`)**.** The Racket expression

```
(lambda (x1 ... xn)
  body)
```

is a **`lambda` expression**. Its value is the function with parameters `x1`, …, `xn` whose value at its arguments is the value of `body`, found as in Definition 7. The function has no name. The body may use any name defined where the `lambda` expression appears, as the body of the `lambda` in `pair-with` (Exercise 7) uses `y`.
The definition `(define (f x) body)` is shorthand for `(define f (lambda (x) body))`: it gives the name `f` to the function the `lambda` expression makes.
Video: Jerry Cain uses `lambda` with `map` in Scheme, the language Racket grew out of, in Stanford CS 107 Lecture 22 (Cain 2008).

Remark 4 (Where the name comes from). The word `lambda` comes from the lambda calculus, a model of computation in which a function is written $\lambda x.\, e$, the function that takes x to the value of the expression e. The lambda calculus has only three features: functions, applying a function to an argument, and variables. It has no numbers, strings or loops (Crichton 2019). Applying $\lambda x.\, e$ to an argument replaces each x in e by that argument, as in evaluation by substitution (Example 1). The function in Exercise 7 that squares its argument is $\lambda x.\, x \cdot x$.
Media: Will Crichton's notes on the lambda calculus, from Stanford's CS 242, *Programming Languages*, go further (Crichton 2019).

Example 4 (A function that returns a function)**.** The function `adder` takes a number `d` and returns a function:

```
(define (adder d)
  (lambda (x) (+ x d)))
```

`(adder 3)` returns the function $x \mapsto x + 3$, which still knows that `d` is 3 after `adder` has returned. So `adder` is a higher-order function (Definition 13), and its value can be used anywhere a function can:

- applied to an argument;
- passed to `twice` from Exercise 5;
- passed to `map` from Example 3.

```
((adder 3) 4)
; 7
(twice (adder 3) 10)
; 16
(map (adder 1) (list 5 7 9))
; '(6 8 10)
```

Definition 15 (`let`)**.** The Racket expression

```
(let ([name1 e1]
      ...
      [namen en])
  body)
```

evaluates `e1`, …, `en`, gives each value its name, and returns the value of `body`, which can use those names. The names last only inside `body`.

A `let` expression is shorthand for applying a `lambda` expression (Definition 14): it has the same value as `((lambda (name1 ... namen) body) e1 ... en)`. So `(let ([s (+ 1 2)]) (* s s))` and `((lambda (s) (* s s)) (+ 1 2))` both return 9.

Video: Cain uses `let` to save a result that would otherwise be computed twice, and shows how `let` becomes a `lambda`, in CS 107 Lecture 22 (Cain 2008).

Example 5 (Naming a value used twice)**.** The solutions of $ax^2 + bx + c = 0$ with $a \neq 0$ and $b^2 - 4ac \geq 0$ are

$$x = \frac{-b \pm s}{2a}, \qquad \text{where } s = \sqrt{b^2 - 4ac}.$$

Both solutions use s, so a `let` names it and computes it once:

```
(define (roots a b c)
  (let ([s (sqrt (- (* b b) (* 4 a c)))])
    (list (/ (+ (- b) s) (* 2 a))
          (/ (- (- b) s) (* 2 a)))))
```

Since $x^2 - 3x + 2 = (x - 1)(x - 2)$, the roots of $x^2 - 3x + 2$ are 2 and 1:

```
(roots 1 -3 2)
; '(2 1)
```

`let` does not change a variable: inside the body, `s` stands for one value, as a name does in algebra. So `roots` is a pure function (Definition 11).

Remark 5 (Scheme and Racket). Stanford's CS 107, *Programming Paradigms*, taught Scheme in its Spring 2008 offering, and its lectures are free to watch (Cain 2008). Racket grew out of Scheme, and most of the Scheme code in those lectures runs in Racket with few changes:

- Scheme's `car` and `cdr` return the first element of a list and the list of the other elements. Racket accepts them too, but this course uses their clearer names `first` and `rest`.
- A Scheme file needs no first line. A Racket file starts with `#lang racket`.
- Racket writes the cases of a `cond` in square brackets, `[test a]`, where Scheme uses parentheses. Racket accepts both.
- CS 107 typed its code into an interpreter called Kawa. This course uses DrRacket.

Video: Jerry Cain introduces Scheme, its lists and its first recursive functions in CS 107 Lecture 19 (Cain 2008).

Remark 6 (Further reading).

- Jerry Cain's CS 107 handouts on Scheme, on the course page, work many more examples, among them flattening nested lists and listing permutations; his lectures 19 to 23 (Cain 2008) cover the same ground.
- Will Crichton's CS 242 notes on functional basics and algebraic data types build functional programs on types, in the language OCaml (Crichton 2019).
- Stanford's CS 99 teaches Lean 4, a functional programming language in which a computer checks proofs about the programs (Aniva and Mohamed 2025).
- The textbook *How to Design Programs* teaches a step-by-step method for writing functional programs.
- Thomas Mailund's *Functional Programming in R 4* (Apress) applies functional programming to statistical programming in R.

References

Aniva, Leni, and Abdalrhman Mohamed. 2025. *CS 99: Functional Programming and Theorem Proving in Lean 4*. Stanford University. https://web.stanford.edu/class/cs99/.

Cain, Jerry. 2008. *Programming Paradigms (Stanford CS107), Lectures 19–23*. Stanford Engineering Everywhere. https://see.stanford.edu/Course/CS107.

Crichton, Will. 2019. *CS 242: Programming Languages, Fall 2019*. Stanford University. https://stanford-cs242.github.io/f19/.

Wikipedia contributors. 2026a. *Expression (Mathematics) — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Expression_(mathematics).

Wikipedia contributors. 2026b. *Function (Mathematics) — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Function_(mathematics).

Wikipedia contributors. 2026c. *Functional Programming — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Functional_programming.

Wikipedia contributors. 2026d. *Higher-Order Function — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Higher-order_function.

Wikipedia contributors. 2026e. *Pure Function — Wikipedia, the Free Encyclopedia*. https://en.wikipedia.org/wiki/Pure_function.