# Introduction to Racket for CS-135

Jared Pincus

Fall 2020

## DrRacket

### Installation

Visit https://download.racket-lang.org/ and select the following settings:

| | |
|---|---|
| Distribution: | **Racket** |
| Platform: | Your computer's operating system. If you are on Mac or Windows, make sure you select the correct 64- or 32-bit setting depending on your computer's specifications. |
| Variant: | **Regular** |

With those settings picked, download the executable and follow the steps in the installer. The process may vary depending on your OS.

After installing, there will be multiple applications installed on your computer, but the one you want to run is *DrRacket*, the Racket IDE in which we'll write and test code.

### Using DrRacket

DrRacket has two main panels, the code ("definitions") editor and the interactions terminal. If either of these panels isn't visible, go to `View > Show Definitions` or `View > Show Interactions`. The interactions terminal can also be revealed by pressing the "Run" button (green triangle) in the upper right corner. This button additionally refreshes the terminal to be updated with the newest code you've written in the code editor. Racket files you write/save/open will display in the code editor. The interactions window allows you to write and evaluate Racket expressions. This is where you can try out test cases for your code.

## Concepts

### Functional Programming (FP)

So far, you've been using the imperative programming paradigm with languages like Python and Java. Imperative programs have an ordered sequence of *statements* which carry out actions to change the state of memory in the computer.

By contast, functional programs are comprised of *expressions* which are evaluated to return immutable data, without affecting the state of memory. The focus in FP is the inputs and outputs of *pure* functions (functions which don't change any state outside of their own scope).

This transition to functional thinking can be difficult. You'll be tempted to have sequential operations performed in a function's body, but you can't in functional programming. Think about this type of programming mathematically. Mathematical functions accept inputs and return an output by combining other functions in particular ways - they don't have a "sequence" of steps.

## Prefix Notation

In arithmetic we usually use *infix notation*, where operators are placed between their arguments. Alternatively we can use *prefix notation*, where operators are placed before their arguments. This is the notation that Racket uses.

In Racket, all function calls, including their arguments, are enclosed by parentheses. When Racket code is evaluated, the innermost parentheses are always evaluated first. And since all function calls are enclosed with parentheses, the order of operations is completely explicit. Here are some examples of the relationship between standard infix notation and Racket's prefix notation:

| Standard Notation | Equivalent Racket Code |
|---|---|
| $1 + 2$ | `(+ 1 2)` |
| $5 - 7$ | `(- 5 7)` |
| $5x + 2y$ | `(+ (* 5 x) (* 2 y))` |
| $\frac{1+\sqrt{5}}{2}$ | `(/ (+ 1 (sqrt 5)) 2)` |
| $p \wedge q$ | `(and p q)` |
| $\neg a \vee b$ | `(or (not a) b)` |

## Racket

We'll be programming in Racket, a functional programming language based on Scheme which is one of the main dialects of Lisp. Throughout the course, we'll interchangeably refer to the language we're using as both Racket and Scheme. More specifically, we'll be coding with a subset of Racket called EOPL. All of the built-in functions in EOPL are listed here: https://docs.racket-lang.org/eopl/index.html (don't get overwhelmed, you won't ever use most of these functions, but learning how to read this documentation will be useful).

If you're wondering why we're using this weird, uncommon language which you never use again in the CS curriculum, here are some justifications:

- Were it not for the use of Scheme in this class, the only functional programming languages used throughout the curriculum would be OCaml and Erlang (for CS students). It's beneficial to be exposed to multiple functional languages, just as you're exposed to multiple object-oriented or imperative languages, especially because OCaml, Erlang, and Scheme are very different implementations of similar principles.
- As a whole, the programming languages in the Lisp family are used all over the place in computer science, and said family is intimately tied to both the historical and mathematical foundations of CS.
- The purity/uniformity of Scheme's syntax, with only functions and parameters, is useful in the context of the material in this class. It makes mathematical and structural analysis of Scheme code very straightforward.
- Setting up and running Racket is much simpler and less error-prone than other similar languages.

# Programming in Racket

## File Format

- Racket files (`.rkt`) are comprised of a set of function definitions which can be arranged in any order.

- At the top of every Racket file goes the declaration of which Racket module is in use. By default this is `#lang racket`, but we'll always change it to be `#lang eopl` to only enable the EOPL subset of the language.

## Syntax

- Line comments are written with `;`. Everything after a semicolon in a line will be commented out. Usually full-line comments start with two semicolons as a convention.
- Block comments begin with `#|` and end with `|#`. Everything in-between, on one or more lines, will be commented out.
- Unlike in Python, whitespace (tabs and line breaks) are ignored in Scheme. All that matters for scoping is the placement of parentheses. Whitespace merely improves readability.
- Parameter and function names can contain (pretty much) any symbols. That's why you'll often see function names with hyphens or question marks in them.

## Arithmetic

Because all parens are explicit in Scheme, order of operations is just based on how operations are nested, rather than conventional PEMDAS as used with infix notation. Here is an incomplete list of some common arithmetic functions in Racket:

**Common arithmetic operators**

| Function | Argument Type | Output |
|---|---|---|
| `(+ x1 x2 ... xn)` | numerical | $x_1 + x_2 + \cdots + x_n$ |
| `(- x1 x2 ... xn)` | numerical | $x_1 - (x_2 + \cdots + x_n)$ |
| `(* x1 x2 ... xn)` | numerical | $x_1 \times x_2 \times \cdots \times x_n$ |
| `(/ x1 x2 ... xn)` | numerical | $x_1/(x_2 \times \cdots \times x_n)$ |
| `(max x1 x2 ... xn)` | numerical | $\max(x_1, x_2, \ldots, x_n)$ |
| `(expt x y)` | numerical | $x^y$ |
| `(abs x)` | numerical | $\lvert x \rvert$ |

## Boolean Logic

The values of true and false are represented by the literals `#t` and `#f` respectively. Here are some predicates (functions which return a boolean) built into Racket which come in handy:

**Common predicate operators**

| Function | Argument Type | Output |
|---|---|---|
| `(and p1 p2 ... pn)` | boolean | $p_1 \wedge p_2 \wedge \cdots \wedge p_n$ |
| `(or p1 p2 ... pn)` | boolean | $p_1 \vee p_2 \vee \cdots \vee p_n$ |
| `(not p)` | boolean | $\neg p$ |
| `(= x1 x2 ... xn)` | numerical | $x_1 \stackrel{?}{=} x2 \stackrel{?}{=} \cdots \stackrel{?}{=} x_n$ |
| `(equal? v1 v2)` | any datatype | $v_1 \stackrel{?}{=} v_2$ |
| `(< x1 x2)` | numerical | $x_1 < x_2$ |
| `(> x1 x2)` | numerical | $x_1 > x_2$ |
| `(<= x1 x2)` | numerical | $x_1 \leq x_2$ |
| `(>= x1 x2)` | numerical | $x_1 \geq x_2$ |
| `(zero? x)` | numerical | $x \stackrel{?}{=} 0$ |

## Function Definitions

Functions definitions take the following form:

```
(define (func-name arg-1 arg-2 ... arg-n) func-body)
```

The function above is called with `(func-name input-1 input-2 ... input-n)` and returns the result of evaluating `func-body` with inputs 1 through $n$ substituted in for arguments 1 through $n$. Just like in any other context, if `func-body` consists of a function call (as it almost always does) it will have to be enclosed with parens.

Suppose in the code editor you defined the function `my-adder` which adds two numbers:

```
(define (my-adder op1 op2) (+ op1 op2))
```

To test out this function with, for instance, inputs 2 and 5, you would press "Run" to update the interactions terminal, input `(my-adder 2 5)` after the ">", and press enter to see the result of evaluating this expression: `7`.