s-expressions

\((op \ arg_1 \ arg_2 \ ... \ arg_n)\)

Rules:

- the operation always comes first
- its arguments (if there are any) follow the operation
- no commas between arguments
- everything goes between parentheses

Common mistakes:

- forgetting parentheses
- rational vs. integer division (/ vs. quotient)
- equality (= vs. equal?)

Examples / Questions:

“Variables”

They’re called variables, but we won’t vary them (i.e., their values are constant).

\(\text{(let* } ([var_1 \ expr_1] \ ... \ [var_n \ expr_n]) \ body-expr)\)

Bindings: an “assignment”. We say: “bind a value to a variable” or “bind a variable to a value”.

The body is the scope of the variables that are bound in the first part of the let*.

Examples / Questions:
Conditionals

(if conditional-expr
   true-expr
   false-expr)

(cond [condition₁ expr₁]
       ...
       [conditionₙ exprₙ]
       [else else-expr])

If you have more than one condition, use cond; otherwise use if.

Examples / Questions:

Functions

(define (function-name parameter₁ ... parameterₙ)
   body-expr)

Questions:
Testing

Test-driven development (TDD): write tests first!
In Racket, we use the rackunit library, like so:

(require rackunit)

Racket Lists

Creating lists in Racket

empty constructs an empty list
(list <value1> ... <valueN>) constructs a list with the given arguments as elements
(cons <value> <list>) given an element and another list, constructs a new list by prepending the element to the provided list. (Careful! If the second element is not a list, then cons does not construct a new list.)

Accessing lists in Racket

(empty? <value>) returns true if the argument is an empty list.
(first <list>) returns the first element of the list
(rest <list>) returns a list that contains every element except the first element of the list.

Inductive data types & recursive operations

Induction is a “building up” process: An inductive data type is one that can be built by starting with the smallest possible instance of that data type, then incrementally building larger instances of it. Racket lists (which are linked lists) are an example of an inductive data type.

Recursion is a “breaking down” process: A recursive function is one breaks a larger problem into a smaller version of that problem that is easier to solve.

If we define a recursive function over an inductive data type, our code usually follows a pattern. For example, here is a recursive function to compute the length of a Racket list:

(define (len L)
  (if (empty? L) ; base case: is this the smallest list?
    0 ; If so, our job is easy.
    (+ 1 (len (rest L)))) ; recursive case: if it’s not the smallest list, solve the problem on a smaller list and use the results to compute the size of this list.
)

Next time: even more recursive operations!