Rules of Processing
Short-Circuiting Rules
The last rule of evaluation (for now)

(define (f x) x)
(f 4)
The goal of the proc-app-expr rule
Subtle questions

“Didn’t you say we couldn’t define things twice? Isn’t that what the proc-app-expr rule is doing?”
- Yes. But I used “define” very specifically: the rules say that if, *in a definition*, the name is already bound to a value, it’s an error
  - There are other ways that names get bound to values (i.e., the proc-app-expr rule!)
  - No, that’s not what the proc-app-expr rule is doing.

“How are values in Racket stored? Directly, or by reference?”
- *Not* a question most of you should be ready to ask, or to even understand!
- Answer: It’s not specified, because *it doesn’t matter.*
  - That lets you, the Racket implementor, make whatever choice is easiest for you!
  - The power of abstraction: ignoring the parts that don’t matter *gives you more power.*
A new bit of syntax: cond
Cond-expression structure
Cond example

(define x 5)

(cond
  [(negative? x) “negative“]
  [(positive? x) “positive“]
  [(zero? x) “zero“])

The first test produces “false”; we move on.
The second test produces “true”; we evaluate the result to get “positive”.
We’re done; we never evaluate the third test.

If we’d defined x to be 0, we’d have gotten “zero”.

Base-pairs

• In biology, there are things called “bases” and named A, T, C, G
• They come in “complementary pairs”: A and T are complements, C and G are complements
• We’ll represent these with strings, “A”, ”T”, “C”, “G”.
• Write a procedure “complement” that consumes a base and produces the complementary base
;; Data definition
;; string: "abs", "T"
;; Data definition
;; string: "abs", "T"
;;
;; complement: string -> string
;;
;; Data definition
;; string: "abs", "T"
;;
;; complement: string -> string
;;

(define (complement base) ...)
;; Data definition
;; string: "abs", "T"
;;
;; complement: string -> string
;;
;; input:
;;   base, a string, which must be one of "A", "T", "C", "G"
;; output:
;;   the complementary base, one of "A", "T", "C", "G"
(define (complement base) ...)
;; Data definition
;; string: "abs", "T"
;;
;; complement: string -> string
;;
;; input:
;;   base, a string, which must be one of "A", "T", "C", "G"
;; output:
;;   the complementary base, one of "A", "T", "C", "G"
(define (complement base) ...)

(check-expect (complement "A") "T")
(check-expect (complement "T") "A")
(check-expect (complement "C") "G")
(check-expect (complement "G") "C")
;; Data definition
;; string: "abs", "T"

;; complement: string -> string  

;; input:
;; base, a string, which must be one of "A", "T", "C", "G"

;; output:
;; the complementary base, one of "A", "T", "C", "G"

define (complement base)
  (cond
   [(string=? base "A") "T"]
   [(string=? base "T") "A"]
   [(string=? base "C") "G"]
   [(string=? base "G") "C"]))

(check-expect (complement "A") "T")
(check-expect (complement "T") "A")
(check-expect (complement "C") "G")
(check-expect (complement "G") "C")
General pattern here

• The structure of the program follows the structure of the data
• Repeated principle throughout CS17
• Almost always a good start when writing any program

;; input:
;;  base, a string, which must be one of "A", "T", "C", "G"
;; ...
(define (complement base)
  (cond
    [(string=? base "A") "T"]
    [(string=? base "T") "A"]
    [(string=? base "C") "G"]
    [(string=? base "G") "C"]))
;; Data definition
;; string: "abs", "T"
;;
;; complement: string -> string
;;
;; input:
;;  base, a string, which must be one of "A", "T", "C", "G"
;; output:
;;  the complementary base, one of "A", "T", "C", "G"
(define (complement base)
  (cond
    [(string=? base "A") "T"]
    [(string=? base "T") "A"]
    [(string=? base "C") "G"]
    [(string=? base "G") "C"]))

(definition check-expect (complement "A") "T")
(definition check-expect (complement "T") "A")
(definition check-expect (complement "C") "G")
(definition check-expect (complement "G") "C")

What should this program produce when given the input “B”? 
Evaluation examples

i.
(define (add1 x) (+ x 1))
(+ (add1 3) (add1 4))