CS135 L03 — Conditionals, Symbols, Testing & Contracts

Luke Lu • 2025-09-16

Piecewise Logic with `cond`

Use `cond` for multiple, ordered conditions with a final `else`.

```
(define (clip01 x)
  (cond [(< x 0) 0]
        [(> x 1) 1]
        [else x]))
```

Substitution walk-through

```
(clip01 -2)
(cond [(< -2 0) 0] [(> -2 1) 1] [else -2])
; first question is true → take its answer
0
```

Else is a catch-all

```
(cond [(< x 0) 'neg]
      [(= x 0) 'zero]
      [else 'pos])
```

> If you are nesting `if`, try `cond` with clear [question ⟶ answer] pairs; it is easier to read and test.

Symbols (Sym) — Use Names, Not Magic Numbers

Symbols are self-documenting values written with a leading quote.

```
(symbol? 'jacket)         ; #true
(symbol=? 'jacket 'coat)  ; #false
```

Example — What to wear?

```
(define (cold? t) (< t 8))
(define (cool? t) (and (< t 16) (not (cold? t))))

;; what-to-wear: Num -> (anyof 'jacket 'sweater 'shirt)
(define (what-to-wear t)
  (cond [(cold? t) 'jacket]
        [(cool? t) 'sweater]
        [else 'shirt]))
```

Testing Strategy — Exact vs Inexact, and Coverage

Write tests early; include boundary/extreme cases.

```
(require rackunit)

; exact comparison
(check-expect (what-to-wear 8) 'sweater)
```

```
; inexact comparison (tolerance)
(check-within (sqrt 2) 1.41421356 1e-6)
```

Branch coverage checklist for a `cond` with two non-else questions:

- Case 1: first true → second not evaluated.
- Case 2: first false, second true.
- Case 3: both false → `else`.

> 💡 **Tip** — When you add a new branch, add at least one new test that **forces** it to run.

Contracts & Requires

Contracts document input/output **kinds** (checked by course tools/graders). Use **Requires** for preconditions not captured by the kind.

```
;; bmi: Num Num -> Num
;; Requires: height > 0
;; Purpose:  Compute body-mass index kg/m^2
(define (bmi mass height)
  (/ mass (* height height)))

(require rackunit)
(check-within (bmi 68 1.75) 22.2 0.1)
```

Restricted symbolic outputs with `anyof`:

```
;; category: Num -> (anyof 'under 'normal 'over 'obese)
(define (category b)
  (cond [(< b 18.5) 'under]
        [(< b 25)   'normal]
        [(< b 30)   'over]
        [else       'obese]))
```

> ⚠️ **Warning** — **Requires** clauses are promises the **caller** must keep. Design tests near boundaries (if allowed) to check behavior is sensible.

Illustrated Example — Safe Square Root

We will design `safe-sqrt` that returns `0` for negative inputs and `sqrt(x)` otherwise.

```
;; safe-sqrt: Num -> Num
;; Purpose: Return 0 if x < 0, else sqrt(x).
(define (safe-sqrt x)
  (cond [(< x 0) 0]
        [else (sqrt x)]))

(require rackunit)
(check-expect (safe-sqrt -9) 0)
(check-within (safe-sqrt 2) 1.41421356 1e-6)
(check-expect (safe-sqrt 0) 0)
```

Submission Checklist (Post-L03)

- For each function: **Purpose**, **Contract**, **Definition**, **Tests** (with branch coverage).

- Label parts clearly; follow the assignment's allowed-constructs list.
- Keep your file clean: header block, reasonable line lengths, organized tests.

Extra Practice

Write and test each function with boundary cases:

```
;; clip-to: Num Num Num -> Num
;; Purpose: clip x into [lo, hi].
(define (clip-to lo hi x)
  (cond [(< x lo) lo]
        [(> x hi) hi]
        [else x]))

(require rackunit)
(check-expect (clip-to 0 1 -0.1) 0)
(check-expect (clip-to 0 1  1.5) 1)
(check-expect (clip-to 0 1  0.4) 0.4)

;; wear-at: Num -> (anyof 'jacket 'sweater 'shirt)
(define (wear-at t)
  (cond [(< t 8)  'jacket]
        [(< t 16) 'sweater]
        [else      'shirt]))
(check-expect (wear-at  5) 'jacket)
(check-expect (wear-at 12) 'sweater)
(check-expect (wear-at 20) 'shirt)
```