CS135 L04 — Recursion

Luke Lu • 2025-09-18

From "..." to Recursion

Goal: Formalize computations that math often writes with ellipses ("...").

Example — Sum to n Let f(n) be the sum of natural numbers up to n. Observe:

- f(n) = f(n − 1) + n
- Base: f(0) = 0

Direct Racket translation (Nat recursion with zero?/sub1):

```
;; sum-to: Nat -> Nat
(define (sum-to n)
  (cond [(zero? n) 0]
        [else (+ n (sum-to (sub1 n)))]))

(require rackunit)
(check-expect (sum-to 4) 10)
(check-expect (sum-to 8) 36)
(check-expect (sum-to 13) 91)
(check-expect (sum-to 0) 0)
```

Key ideas:

- A **recursive function** calls itself with a "smaller" argument until a **base case**.
- For Naturals in this course: "smaller" = `sub1`, base test = `zero?`.

> **The "Tripod" of Computation (Turing completeness in our model):** 1) Arbitrarily large numbers • 2) Conditionals • 3) Recursion.

Rules for Recursion (version 1)

1) **Change exactly one argument** closer to termination on the recursive call. 2) For Naturals: **use `sub1`** and test termination with **`zero?`**.

Template for Natural-number recursion:

```
(define (natural-template n)
  (cond
    [(zero? n) ...]          ; base case result
    [else (... n ... (natural-template (sub1 n)))]) ; use n and recur on (sub1 n)
)
```

> 💡 **Tip — Three properties to watch:**
>
> - **Termination:** Guaranteed by shrinking the Nat with `sub1` until `zero?`.
> - **Correctness:** In CS135 we lean on tests (you could also prove by induction).
> - **Efficiency:** Measured by **stepper** (count substitution steps).

Names & Scope

- **Scope** = where an identifier (function/parameter/constant) has effect.

- Two kinds for now: **global** and **function** scope.
- The **smallest enclosing scope** takes priority.
- **Duplicate identifiers in same scope → error**:

```
(define f 3)
(define (f x) (sqr x)) ; error: f already defined as a value
```

- DrRacket has **scope tools** to visualize where names refer.

Our 5-Step Design Pattern

1) **Purpose**: Write what the function produces/consumes. 2) **Contract & Header**: Types (kinds) and parameter names. 3) **Examples/Tests**: Work by hand; include boundary cases. 4) **Body from Template**: Start with the recursion template, fill in specifics. 5) **More Tests**: Add any missing edge/coverage tests.

Worked Example — Factorial

Math idea: $n! = n \cdot (n-1)!$ with base $0! = 1$ (by convention).

Definition (Nat recursion):

```
;; factorial: Nat -> Nat
;; Produce n!
(define (factorial n)
  (cond
    [(zero? n) 1]
    [else (* n (factorial (sub1 n)))]))

(require rackunit)
(check-expect (factorial 3) 6)
(check-expect (factorial 5) 120)
(check-expect (factorial 7) 5040)
(check-expect (factorial 0) 1)
(check-expect (factorial 1) 1)
```

Variant with a shorter name:

```
;; !: Nat -> Nat
(define (! n)
  (cond
    [(zero? n) 1]
    [else (* n (! (sub1 n)))]))
```

> ⚠ **Warning** — Common mistakes:
> - Changing **multiple** arguments on the recursive call (breaks our v1 rules).
> - Missing or incorrect **base case** (non-terminating or wrong result).
> - Forgetting to **use n in the step** (e.g., summing/multiplying the wrong thing).