CS135 L06 — Recursion on Lists

Your Name • 2025-09-25

CS135 L06 — Recursion on Lists

Prepared by Your Name

L06.0 — Buying apples (motivation)

- Model a grocery list as a recursive list of `Food`.
- Data definition pattern for lists:

```
;; a Food is (anyof 'apple 'bread 'eggs 'milk)
;; a (listof Food) is one of:
;;  * empty
;;  * (cons Food (listof Food))
```

- Count apples:

```
;; count-apples: (listof Food) -> Nat
(define (count-apples groceries)
  (cond [(empty? groceries) 0]
        [(symbol=? 'apple (first groceries))
         (add1 (count-apples (rest groceries)))]
        [else (count-apples (rest groceries))]))
```

:contentReference[oaicite:7]{index=7}

L06.1 — Templates & Rules (v2)

- General list template:

```
(define (listof-X-template lox)
  (cond [(empty? lox) ...]
        [... ...]
        [else (... (first lox)
                   (listof-X-template (rest lox)))]))
```

- Four planning questions: base result, handle first, result on rest, how to combine.
- **Rules for recursion (second version):** 1) Change **one** argument closer to termination; others stay fixed. 2) On naturals: recur with `(sub1 n)`, test with `zero?`. 3) On lists: recur with `(rest lst)`, test with `empty?`. :contentReference[oaicite:8]{index=8}

Generalizing: count a target symbol

```
;; count-symbol: Sym (listof Sym) -> Nat
(define (count-symbol target lst)
  (cond [(empty? lst) 0]
        [(symbol=? target (first lst))
         (add1 (count-symbol target (rest lst)))]
        [else (count-symbol target (rest lst))]))
```

- Tests:

```
(check-expect (count-symbol 'apple empty) 0)
(define test0
  (cons 'apple
        (cons 'eggs
```

```
              (cons 'bread
                    (cons 'apple
                          (cons 'milk
                                (cons 'bread empty)))))))
(check-expect (count-symbol 'apple test0) 2)
(check-expect (count-symbol 'fish test0) 0)
```

:contentReference[oaicite:9]{index=9}

L06.2 — Length & list idioms

- Length (don't name it `length` here to avoid clash):

```
;; len: (listof Any) -> Nat
(define (len lst)
  (cond [(empty? lst) 0]
        [else (add1 (len (rest lst)))]))
```

- **List idioms** introduced:
  - **Folding** (reduce to a single value) — `count-symbol`, `len` are folds.
  - **Mapping** (transform each element).
  - **Filtering** (keep elements meeting a condition). :contentReference[oaicite:10]{index=10}

L06.3 — Predicates over lists

- Example: "all positive" predicate with short-circuit on first non-positive.

```
;; all-positive?: (listof Num) -> Bool
(define (all-positive? lst)
  (cond [(empty? lst) true]
        [(<= (first lst) 0) false]
        [else (all-positive? (rest lst))]))
```

- Do not use `member?` / `equal?` yet—write your own membership checks (`contains-symbol?`, then a mixed type version) to reason about **type-appropriate equality**. :contentReference[oaicite:11]{index=11}

L06 — You should know

- How to write recursive list functions using the **template** + **Rules (v2)**.
- How to fold a list; how to build predicates over lists.
- Why some powerful built-ins (e.g., `member?`, `equal?`) are disallowed now. :contentReference[oaicite:12]{index=12}

L06 — Allowed constructs (highlights)

New: `add1 length listof` + Rules for Recursion (v2). Previously allowed: core arithmetic/logic, `cond`, `define`, `cons/first/rest`, `empty/empty?`, tests, symbol/number predicates, etc. Recursion must follow second-version rules. :contentReference[oaicite:13]{index=13}