

CS 135 – L17 - First Class Functions

Luke Lu • 2025-12-17

Filter

Info – filter

```
(filter pred? lst)
```

Filter function is equivalent to keep function.

```
(define (keep pred? lst)
  (cond [(empty? lst) empty]
        [(pred? (first lst)) (cons (first lst) (keep pred? (rest lst)))]
        [else (keep pred? (rest lst))])
        )
      )
```

Time Complexity: $O(n)$

Example Template:

```
(define function lst)
  (local [(define (pred? ...))])
    (filter pred? lst))
```

Producing Functions

Info – Adders

Adders increment the given n by a m (i.e. (add1 n) is an adder with $m = 1$)

```
define (make-adder n)
  (local
    [(define (f m) (+ n m))]
    f))
```

List of Functions

Evaluation of Expression Tree

Approach 1

```
;; translate: Sym -> (list (Num Num -> Num) Num)
(define (translate op)
  (local[(define (nothing x y) 0)
         (define operations (list '+ (list + 0)) (list '* (list * 1))))
         (define (lookup op lst)
           (cond [(empty? lst) (list nothing 0)]
                 [(symbol=? op (first (first lst))) (second(first lst))]
                 [else (lookup op (rest lst))]))
         (lookup op operations)))
```

```

;; apply an operator to a list of arithmetic expressions
;; apply-exp: Sym (listof AExp) -> Num
(define (apply-exp op el)
  (local [(define operation (translate op))]
    (cond [(empty? el) (second operation)]
          [else ((first operation)(eval (first el))
            (apply-exp op (rest el))))]))

```

Approach 2

This requires a slight modification on the expression tree to directly store the operation as the first element of the list

```

;; evaluate an arithmetic expression
;; eval: AExp -> Num
(define (eval exp)
  (cond [(number? exp) exp]
        [else (apply-exp (first exp) (rest exp))]))

;; apply an operator to a list of arithmetic expressions
;; apply-exp: (Num Num -> Num) (listof AExp) -> Num
(define (apply-exp op args)
  (cond [(empty? args) (op)]
        [else (op (eval (first args))
          (apply-exp op (rest args))))]))

```