CSCI 275: Programming Abstractions

Lecture 9: apply & fold right

Spring 2025

Stephen Checkoway, Oberlin College
Slides gratefully borrowed from Molly Q Feldman

Another tool: apply

Motivation

Imagine you have a list of numbers and you want to multiply them all together

We know `(* 3 5 7 -2 8 10)` works but how do we make that into a function if we don't know how many numbers we have ahead of time?

```
(define (product lst)
   ???)
```

We could write a recursive procedure but it'd be great if we could just use the elements in `lst` as the arguments to `*`

Applying a procedure to a list of arguments

`(apply proc lst)`

Applies `proc` to the arguments in `lst` and returns a single value

```
(define (maximum lst)
  (apply max lst))
(maximum '(1 3 4 2)) => (apply max '(1 3 4 2))
                     => (max 1 3 4 2)
                     => 4

(define sum
  (lambda (lst)
    (apply + lst))
(sum '(1 2 3)) => (apply + '(1 2 3)) =>
              (+ 1 2 3) => 6
```

+ in Racket can take any number of arguments

Applying with some fixed arguments

`(apply proc v… lst)`

`apply` takes a variable number of arguments where the final one is a list and applies `proc` to all of those arguments

```
(apply proc 1 2 3 '(4 5 6)) => (proc 1 2 3 4 5 6)
```

You’re working with 3-d vectors stored as 3-element lists—e.g. `'(-5 8 6.2)`. You have a function `(vector-len x y z)` which gives the length of the vector and a list of vectors

`(define vecs '((-5 8 6.2) (1 -2 3) …))`

How do you get a list of the lengths of the vectors?

A. `(map vector-len vecs)`

B. `(apply vector-len vecs)`

C. `(map (λ (v) (apply vector-len v)) vecs)`

D. `(apply (λ (v) (map vector-len v)) vecs)`

Even *more* abstractions, and thus tools in our toolbox

Lots of similarities between functions

(sum lst)

```
(define (sum lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst)
                 (sum (rest lst)))]))
```

(length lst)

```
(define (length lst)
  (cond [(empty? lst) 0]
        [else (+ 1
                 (length (rest lst)))]))
```

(map proc lst)

```
(define (map proc lst)
  (cond [(empty? lst) empty]
        [else (cons (proc (first lst))
                    (map proc (rest lst)))]))
```

Even for functions that don’t immediately look like they fall into the pattern…

`(remove* x lst)`

```
(define (remove* x lst)
  (cond [(empty? lst) empty]
        [(equal? x (first lst)) (remove* x (rest lst))]
        [else (cons (first lst)
                    (remove* x (rest lst)))]))
```

Even for functions that don't immediately look like they fall into the pattern…

`(remove* x lst)`

```
(define (remove* x lst)
  (cond [(empty? lst) empty]
        [(equal? x (first lst)) (remove* x (rest lst))]
        [else (cons (first lst)
                    (remove* x (rest lst)))]))
```

We can rewrite them to look more like the others

```
(define (remove* x lst)
  (cond [(empty? lst) empty]
        [else (if (equal? x (first lst))
                  (remove* x (rest lst))
                  (cons (first lst)
                        (remove* x (rest lst))))]))
```

Some similarities

Basic structure is the same!

```
(define (fun … lst)
   (cond [(empty? lst) base-case]
         [else
          (let ([head (first lst)]
                [result (fun … (rest lst))])
            (combine head result))]))
```

Function	**base-case**	**(combine head result)**
`sum`	`0`	`(+ head result)`
`length`	`0`	`(+ 1 result)`
`map`	`empty`	`(cons (proc head) result)`
`remove*`	`empty`	`(if (equal? x head) result` `(cons head result))`

```
(define (fun lst)
  (cond [(empty? lst) base-case]
        [else (let ([head (first lst)]
                    [result (fun (rest lst))])
                (combine head result))]))
```

lst: list of α

base-case: β

What kind of function is combine?
(input type to output type)

A. combine: $\alpha \times \beta \rightarrow \alpha$

B. combine: $\alpha \times \beta \rightarrow \beta$

C. combine: $\beta \times \alpha \rightarrow \alpha$

D. combine: $\beta \times \alpha \rightarrow \beta$

```
(define (fun lst)
  (cond [(empty? lst) base-case]
        [else (let ([head (first lst)]
                    [result (fun (rest lst))])
                (combine head result))]))
```

lst: list of α

base-case: β

combine: $\alpha \times \beta \rightarrow \beta$

If α = boolean and β = string,

what type is (fun '(#t #f #f))?

A. boolean
B. string
C. boolean → string
D. string → boolean

Next Up

Readings continue, see the course schedule!

Homework 2 is due tonight at 11:59pm via Github

- Commit/Push is free, do it often!

Homework 3 is available