CSCI 275: Programming Abstractions

Lecture 11: Higher Order Wrap-Up

Spring 2025

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

Questions? Comments?

Functional Language of the Week: Python

- Wait, hold on! Python is *not* a functional paradigm language
  - Paradigms are a gray area….
- The transition from Python 2 to Python 3 facilitated significantly better functional programming in Python
  - This was, more or less, due to popular demand
- A (long-running) guide to functional programming in Python https://docs.python.org/3/howto/functional.html#

Functional Language of the Week: Python

```
#filter
x = filter(lambda x: x%3 == 0 and x%5 != 0, [3,6,9,12,15,18,21,24,27,30])
print(list(x)) #gives [3, 6, 9, 12, 18, 21, 24, 27]

#map
a = map(lambda x: x + 1, [1,2,3])
print(list(a)) #gives [2, 3, 4]

#subtle: how do you change how sorted works?
#a lambda!
r = sorted([(1,0), (2,1), (3,2)], key = lambda x: x[1])
print(list(r)) #gives [(1, 0), (2, 1), (3, 2)]
```

foldl

Reminder: Light Switch State Machine

Possible actions: `'up`, `'down`, `'flip`

Possible states: `'on`, `'off`

We want

```
(state-after '(up flip)) => 'off
```

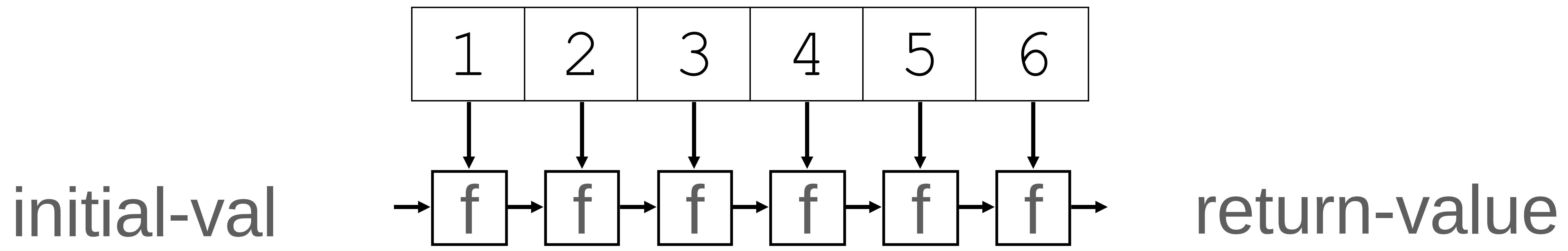
Reminder: Lightswitch, as `foldr` and `foldl`

```
(define state-after
   (lambda (actions)
    (foldr next-state 'off (reverse actions))))
```

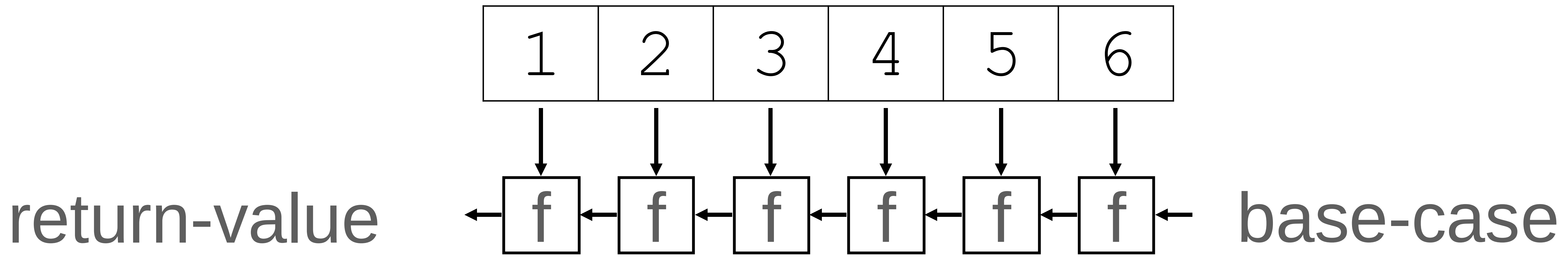
```
(define (state-after-left actions)
  (foldl  next-state 'off actions))
```

foldl vs. foldr

`foldl` combines elements of the list starting with the first (left-most) element



`foldr` combines elements of the list starting with the last (right-most) element



But wait: more “thoughtful” motivation for foldl

Reminder: Tail Recursion and using an “accumulator"

```
(define (fact-a n acc)
  (if (<= n 1)
      acc ; return the accumulator
      (fact-a (sub1 n) (* n acc))))
(define (fact2 n)
  (fact-a n 1))
```

Four things to notice:

- We defined a recursive helper function that takes an additional param
- We provide an initial value for the accumulator in `fact2`'s call to `fact-a`
- The base case returns the accumulator
- `fact-a` is tail-recursive

Product: An Accumulator Pattern

```
(define (product-a lst acc)
  (cond [(empty? lst) acc]
        [else (product-a (rest lst)
                         (* (first lst) acc))]))

(define (product lst)
  (product-a lst 1))
```

Reverse: An Accumulator Pattern

```
(define (reverse-a lst acc)
  (cond [(empty? lst) acc]
        [else (reverse-a (rest lst)
                         (cons (first lst) acc))]))

(define (reverse lst)
  (reverse-a lst empty))
```

Map: An Accumulator Pattern

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

(define (map proc lst)
  (reverse (map-a proc lst empty)))
```

Accumulator Pattern Similarities

Basic structure is the same (rewriting slightly)

```
(define (fun-a lst acc)
  (cond [(empty? lst) acc]
        [else
         (fun-a (rest lst)
                (combine (first lst) acc))]))
(define (fun … lst)
  (fun-a lst initial-val))
```

Function	initial-val	(combine head acc)
product	1	(* head acc)
reverse	empty	(cons head acc)
map	empty	(cons (proc head) acc)

We must reverse the result

Abstraction: fold left

`(foldl combine initial-val lst)`

`combine`: $\alpha \times \beta \to \beta$

`initial-val`: β

`lst`: `list of` α

`foldl`: $(\alpha \times \beta \to \beta) \times \beta \times (\text{list of } \alpha) \to \beta$

Elements of `lst` $= (x_1\ x_2\ \ldots\ x_n)$ and `initial-val` are combined by computing

$z_1 = (\text{combine } x_1 \text{ initial-val})$

$z_2 = (\text{combine } x_2\ z_1)$

$z_3 = (\text{combine } x_3\ z_2)$

$\vdots$

$z_n = (\text{combine } x_n\ z_{n-1})$

product as fold left

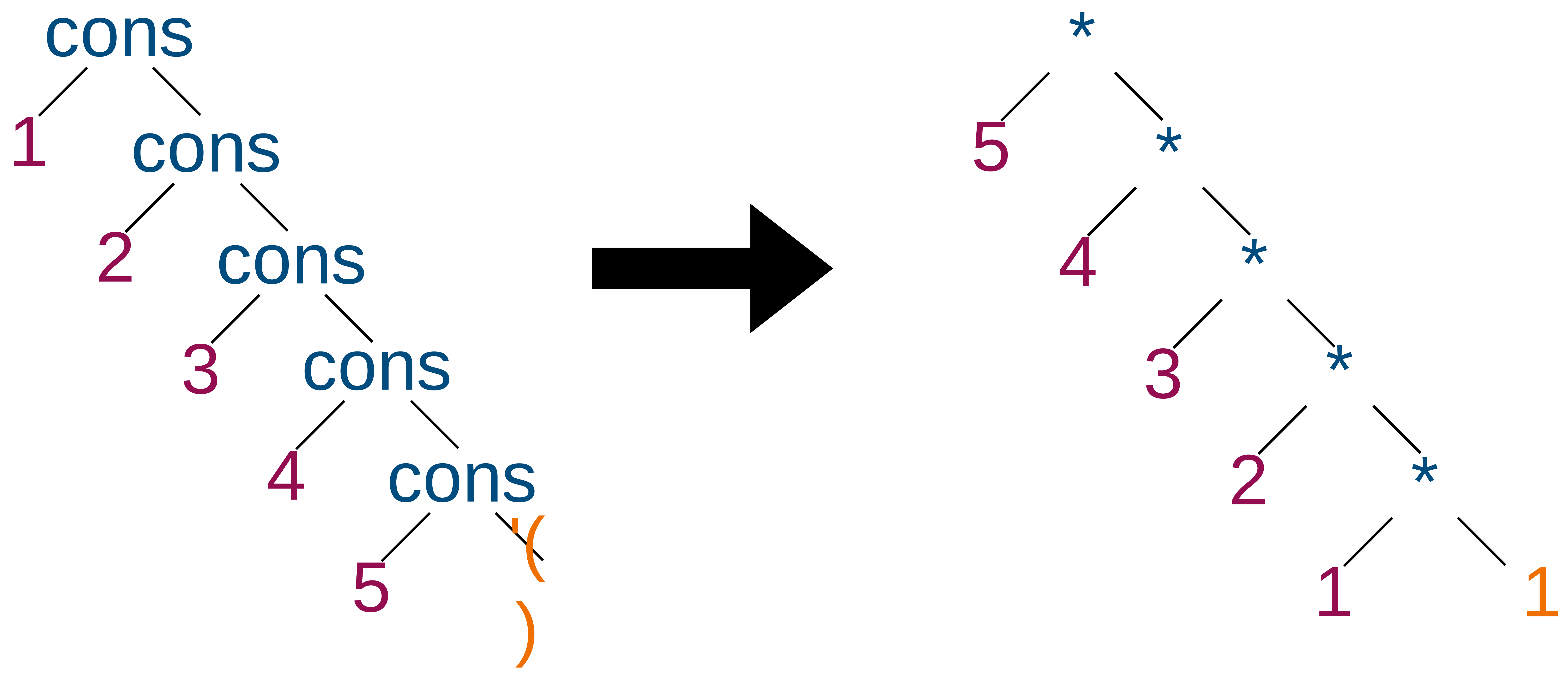
(foldl combine initial-val lst)

```
(define (product lst)
  (foldl * 1 lst))
```

combine: number × number → number

initial-val: number

lst: list of number

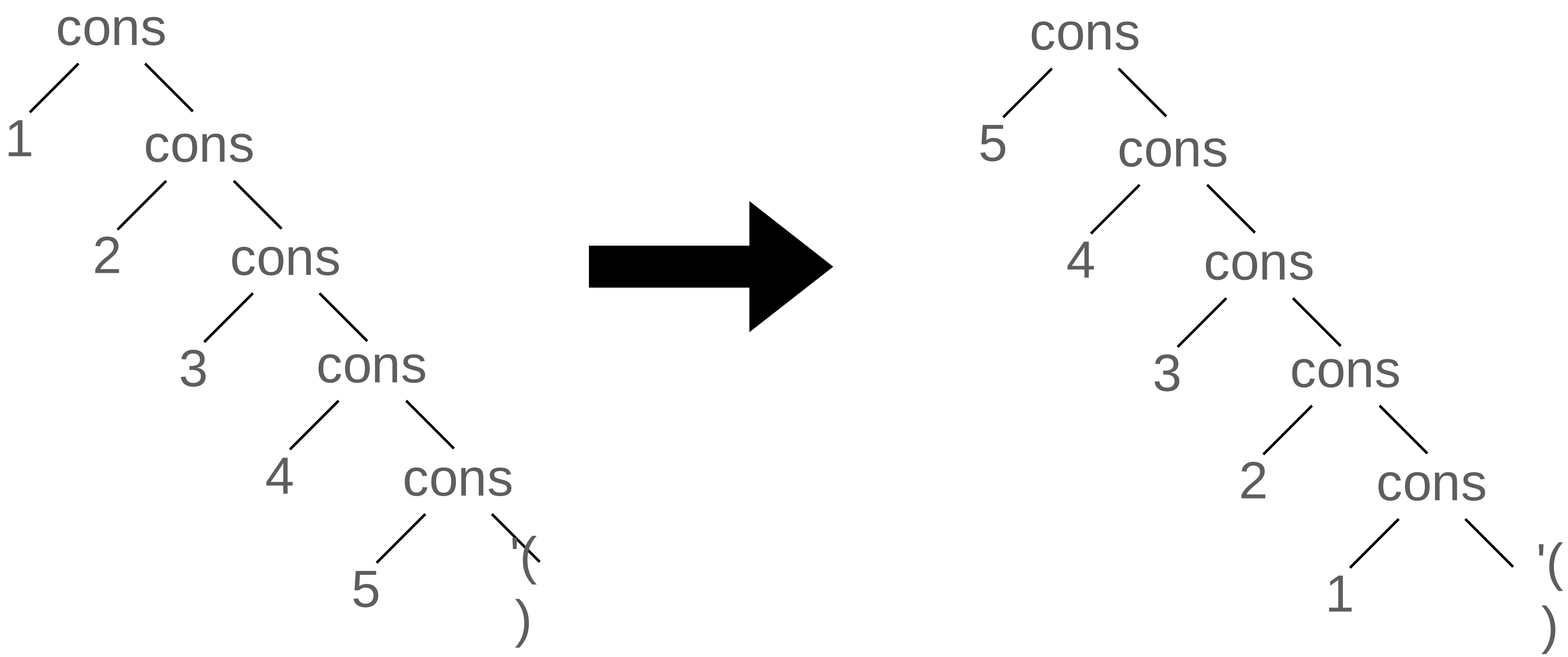


reverse as fold left

`(foldl combine base-case lst)`

```
(define (reverse lst)
  (foldl cons empty lst))
```

combine: α × list of α → list of α
initial-val: list of α
lst: list of α



Which fold to pick?

- "Most of the time", either will work +/- a call to `reverse`
  - Be careful when `combine` has ordering effects
- If the computation makes more sense as a right-to-left computation on the elements of the list, then use `foldr`
- But, most of the time, use `foldl`
  - Lists run left-to-right in Racket world
  - Fold in most other functional contexts assumes `foldl`
  - Tail recursive, and thus more efficient

Aside: foldr can work on lazy, infinite lists

Languages with lazy evaluation can have infinite data structures like infinite-length lists (we'll see something similar in Racket later in the semester)

Fold right can work with infinite lists so long as only a finite portion of the list is required to compute the value

```
nats :: [Int]
nats = [1..] -- An infinite list


sumFirst :: Int -> Int
sumFirst n = foldr (\x acc -> if x <= n
                              then x + acc
                              else 0)
                   (error "this will never be evaluated")
                   nats
```

Variable Argument Procedures

Variable argument procedures

```
(define foo (lambda params body))
```

When `params` is a **list of identifiers** (as we know it thus far!), the identifiers are bound to the values of the procedure's arguments

When `params` is an **identifier** (i.e., not a list), then the identifier is bound to a list of the procedure's arguments

```
(define count-args
  (lambda params
    (length params)))
```

```
(count-args 'a 2 #f) => 3
```

Folks asked about why no parens worked in some previous homeworks, this is why!

Required parameters + variable parameters

```
(define foo (lambda (x y z . params)) body)
```

Separate the required parameters from the list of variable parameters with a period

```
(define drop-2
  (lambda (x y . lst) lst))

(drop-2 1 2 3 4)
```

`x` is bound to `1`
`y` is bound to `2`
`lst` is bound to `'(3 4)`

Review & Practice

There's a standard library procedure `(round x)` that takes a number as input and rounds it to the nearest integer.

If we have a list of numbers `'(1.1 2.9 3.5 4.0)` and we want a list of rounded numbers `'(1.0 3.0 4.0 4.0)`, how can we get that?

A. `(map (round x) '(1.1 2.9 3.5 4.0))`
B. `(map (lambda (x) (round x)) '(1.1 2.9 3.5 4.0))`
C. `(map round '(1.1 2.9 3.5 4.0))`
D. `(round '(1.1 2.9 3.5 4.0))`
E. More than one of the above

Distance of a 2-d point from the origin

Recall that a point (x, y) lies $\sqrt{x^2 + y^2}$ from the origin

Let's make a procedure to compute this

```
(define (distance-from-origin x y)
  (sqrt (+ (* x x) (* y y))))
```

```
(distance-from-origin 3 4) => 5
```

```
(define (distance-from-origin x y)
  (sqrt (+ (* x x) (* y y))))
```

If we have a point

`(define p '(5 -8))` how can we get its distance from the origin?

A. `(distance-from-origin p)`

B. `(apply distance-from-origin p)`

C. `(distance-from-origin (first p) (second p))`

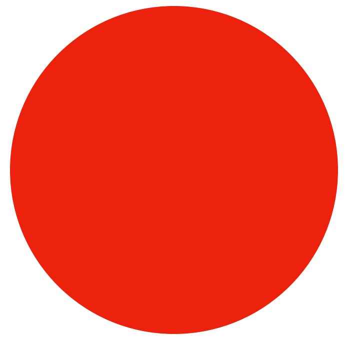
D. More than one of the above

Shapes

Racket library `2htdp/image` has procedures for creating images

```
(require 2htdp/image)

(circle 20 'solid 'red) =>
```

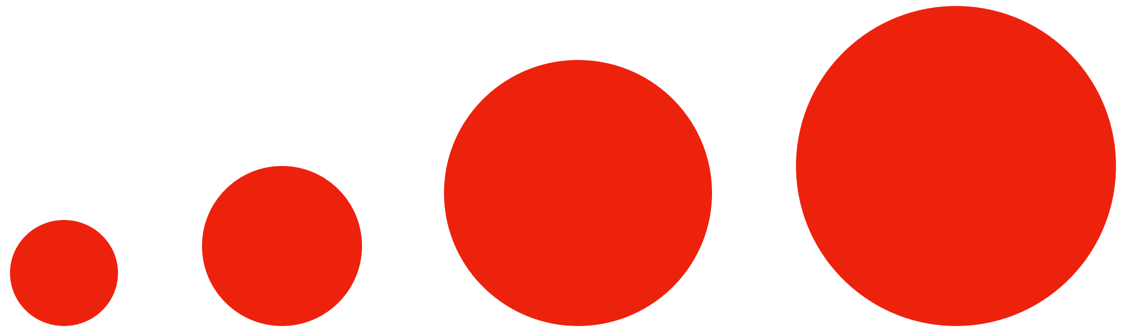


radius

```
(rectangle 50 20 'outline 'blue) =>
```

width height

If we have a list of radii, say `lst` is `'(20 30 50 60)` and we want a list of solid, red circles with those radii, which should we use?

`(_____ lst) => (list`  `)`

A. `(map circle 'solid 'red lst)`

B. `(map (lambda (r) (circle r 'solid 'red)) lst)`

C. `(apply circle 'solid 'red lst)`

D. `(apply (lambda (r) (circle r 'solid 'red)) lst)`

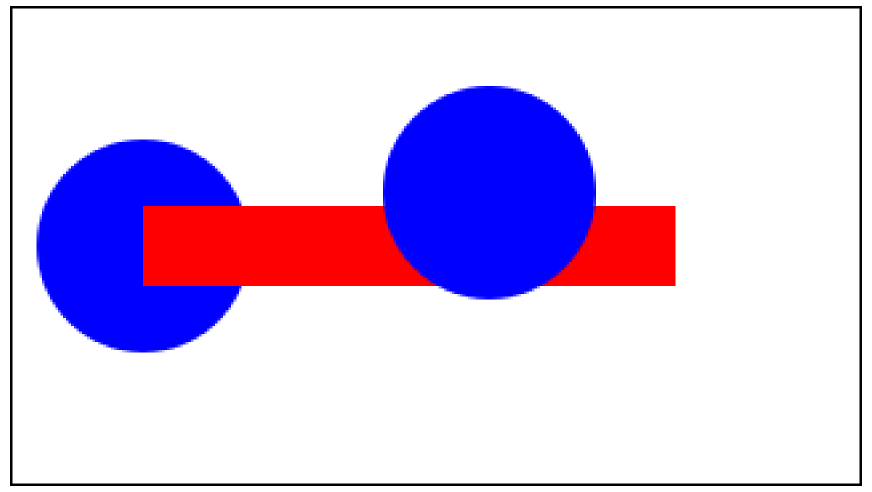
E. `(foldr (lambda (r) (circle r 'solid 'red)) empty lst)`

Combining images

`(empty-scene 320 180)` gives a white rectangle with a black border we can draw on

`(place-image img x y scene)` returns a new image by starting with `scene` and drawing `img` at `(x, y)`

```
(let* ([c (circle 40 'solid 'blue)]
       [r (rectangle 200 30 'solid 'red)]
       [s0 (empty-scene 320 180)]
       [s1 (place-image c 50 90 s0)]
       [s2 (place-image r 150 90 s1)]
       [s3 (place-image c 180 70 s2)])
  s3)
```



Imagine we have a list of 3-element lists `(shape x y)`, e.g., `lst` is the list

```
(list (list (circle 40 'solid 'blue) 50 90)
      (list (rectangle 200 30 'solid 'red) 150 90)
      (list (circle 40 'solid 'purple) 180 70))
```

How would you draw those shapes on a scene at their coordinates?

```
A. (map (lambda (i)
          (place-image (first i) (second i) (third i) scene))
        lst)

B. (apply (lambda (i)
            (place-image (first i) (second i) (third i) scene))
          lst)

C. (foldr (lambda (i s)
            (place-image (first i) (second i) (third i) s))
          scene
          lst)
```

Try out the previous question on your own!