Programming Abstractions

Lecture 10: Fold left

Stephen Checkoway

Review: map

Applies a procedure to each element of a list

α and β are types

```
(map proc lst)
```

- `proc` : $\alpha \to \beta$
- `lst : list of` α
- `map` returns `list of` β

E.g.,

- α = `number`, β = `integer`

  ```
  (map floor '(1.3 2.8 -8.5))
  ```

Review: apply

Applies a procedure the arguments in a list

`(apply proc lst)`

- `proc` : $\alpha_1 \times \alpha_2 \times \cdots \times \alpha_n \to \beta$
- `lst` : `(`α_1 α_2 … α_n`)`
- `apply` returns β

E.g.,

- α_1 = `number`, α_2 = `boolean`, β = `number`

```
(apply (λ (n b) (if b (- n) n))
       '(5 #t))
```

Review: fold right

Folds let us combine all elements of a list

```
(foldr combine initial lst)
```

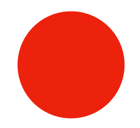
- `combine` : $\alpha \times \beta \rightarrow \beta$
- `initial` : β
- `lst` : list of α
- `foldr` returns β

E.g., α = `string` and β = `number`

```
(foldl (λ (str num) (+ num (string-length str)))
       0
       '("red" "green" "blue"))
```

Shapes

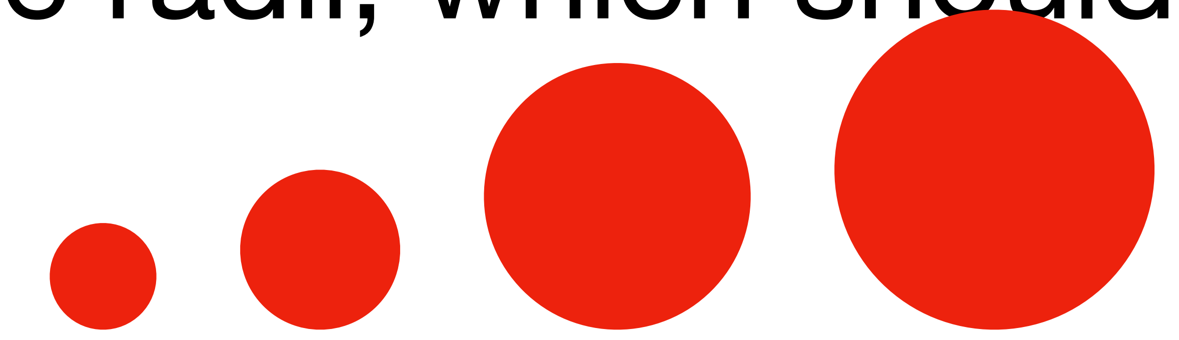
Racket library 2htdp/image has procedures for creating images

```
(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 (λ (r) (circle r 'solid 'red)) lst)`

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

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

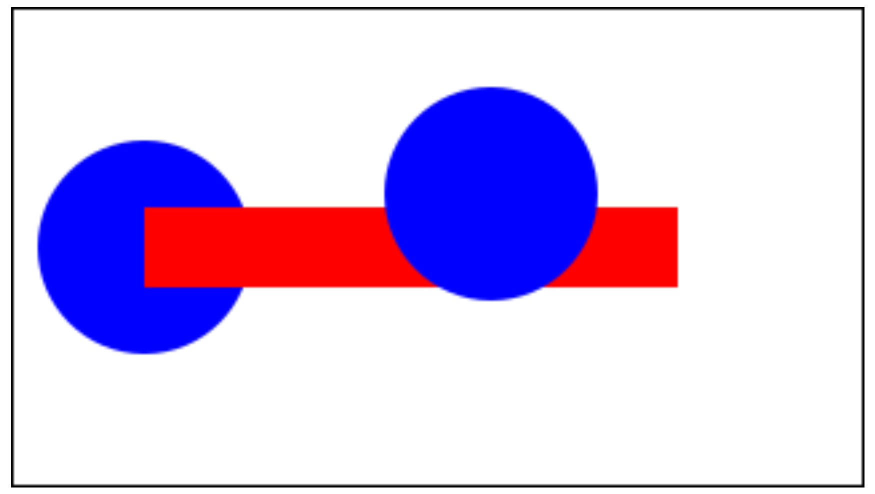
E. `(foldr (λ (r) (circle r 'solid 'red)) empty lst)`

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?

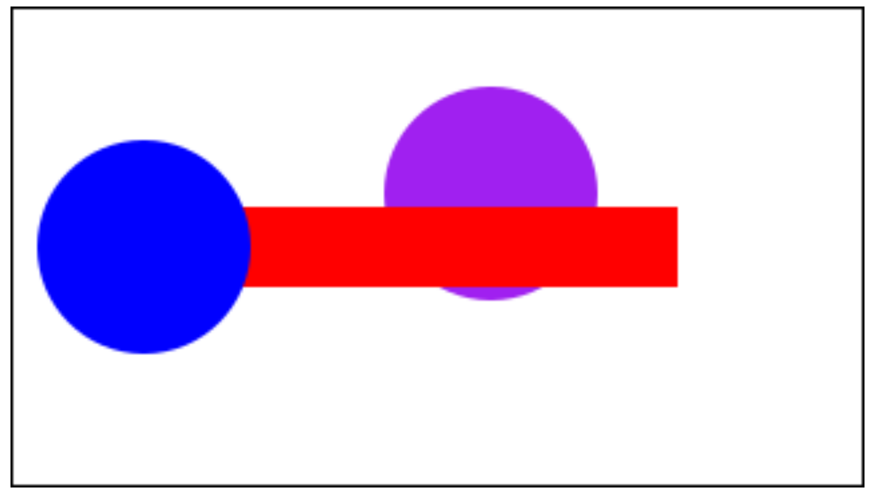
A.
```
(map (λ (i) (place-image (first i) (second i) (third i) scene))
     lst)
```

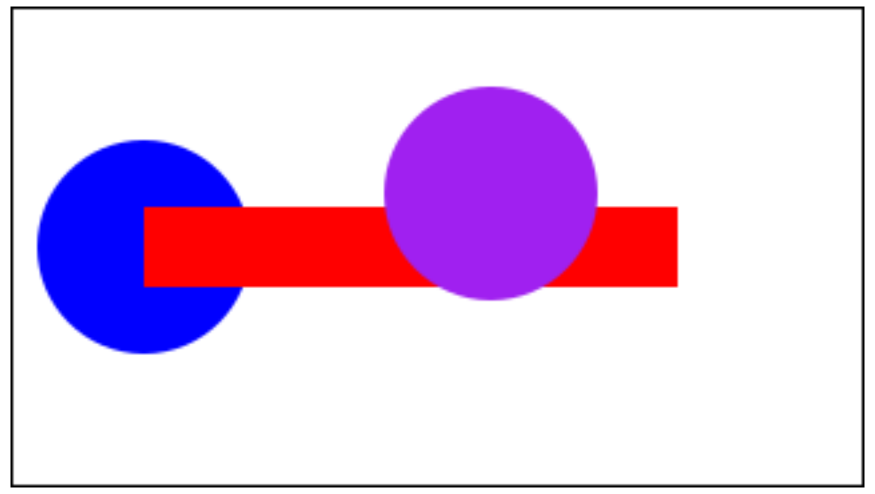
B.
```
(apply (λ (i) (place-image (first i) (second i) (third i) scene))
       lst)
```

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

```
(define lst
  (list (list (circle 40 'solid 'blue) 50 90)
        (list (rectangle 200 30 'solid 'red) 150 90)
        (list (circle 40 'solid 'purple) 180 70)))
(foldr (λ (i s) (place-image (first i) (second i) (third i) s))
       (empty-scene 320 180)
       lst)
```

Which image is drawn by this code?

A. 

B. 

C. There's not enough information to know

Accumulation-passing style similarities

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

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

Accumulation-passing style similarities

```
(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))
```

Accumulation-passing style similarities

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

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

Some 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

`foldr`: $(\alpha \times \beta \to \beta) \times \beta \times (\texttt{list of } \alpha) \to \beta$

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

z_1 = `(combine` x_1 `initial-val)`

z_2 = `(combine` x_2 z_1`)`

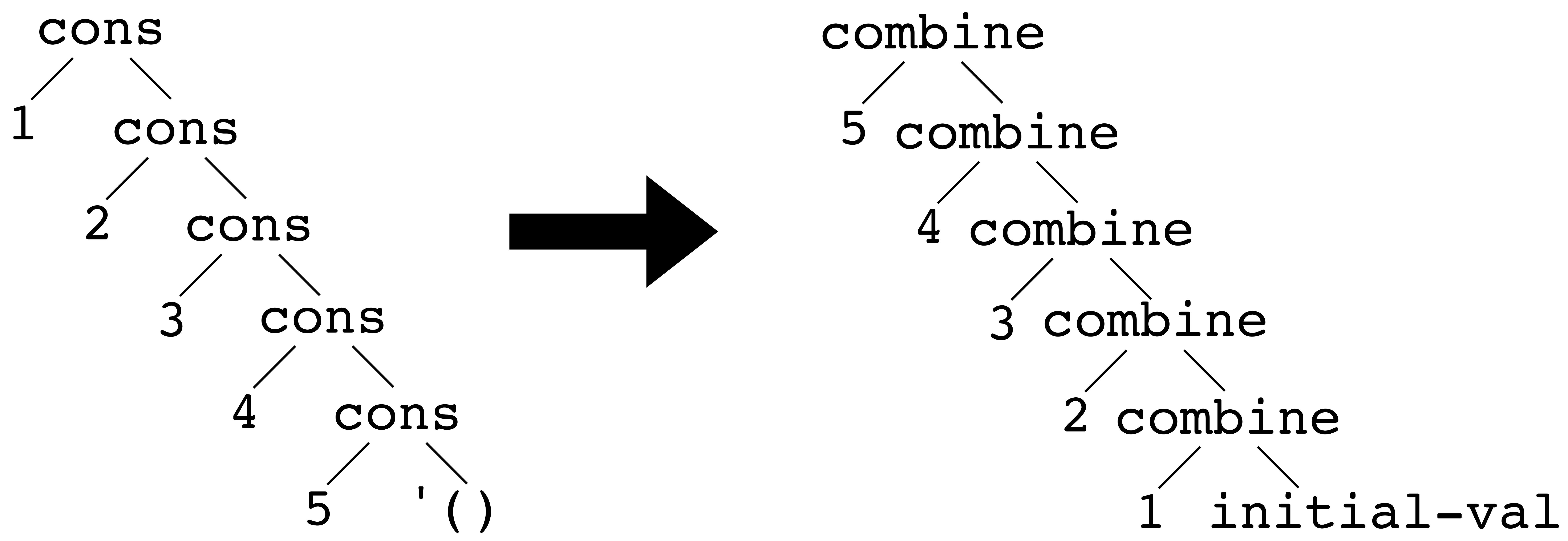
z_3 = `(combine` x_3 z_2`)`

$\vdots$

z_n = `(combine` x_n z_{n-1}`)`

Abstraction `foldl`

```
(foldl combine initial-val lst)
```

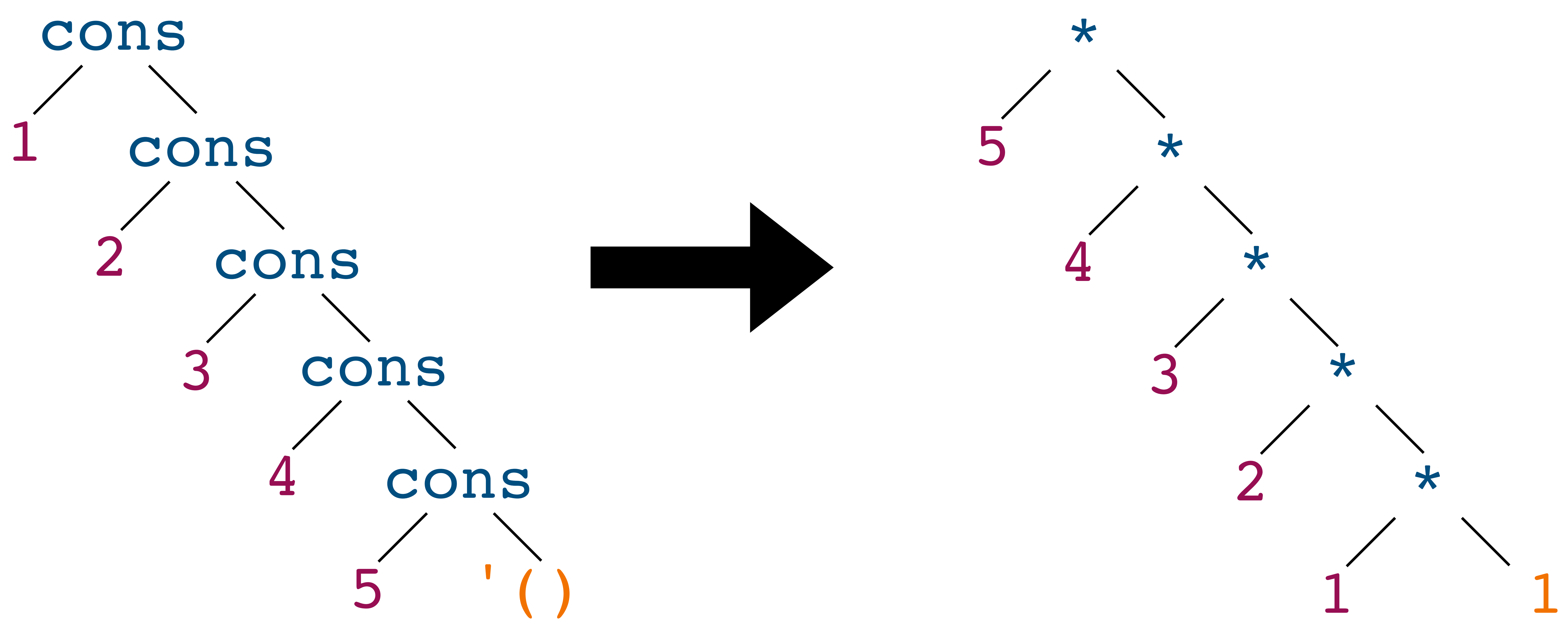


product as fold left

`(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

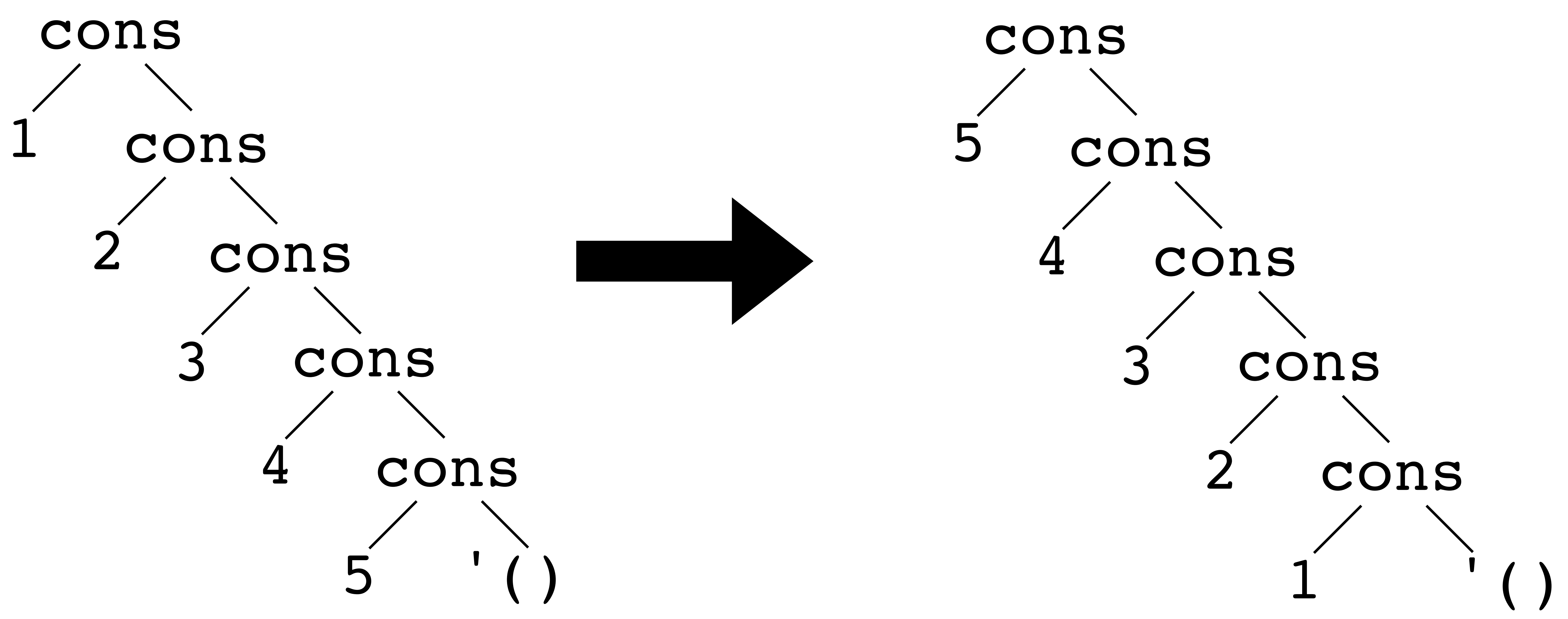
`(foldl combine base-case lst)`

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

combine: α × list of α → list of α

initial-val: list of α

lst: list of α



`combine: α × list of α → list of α`

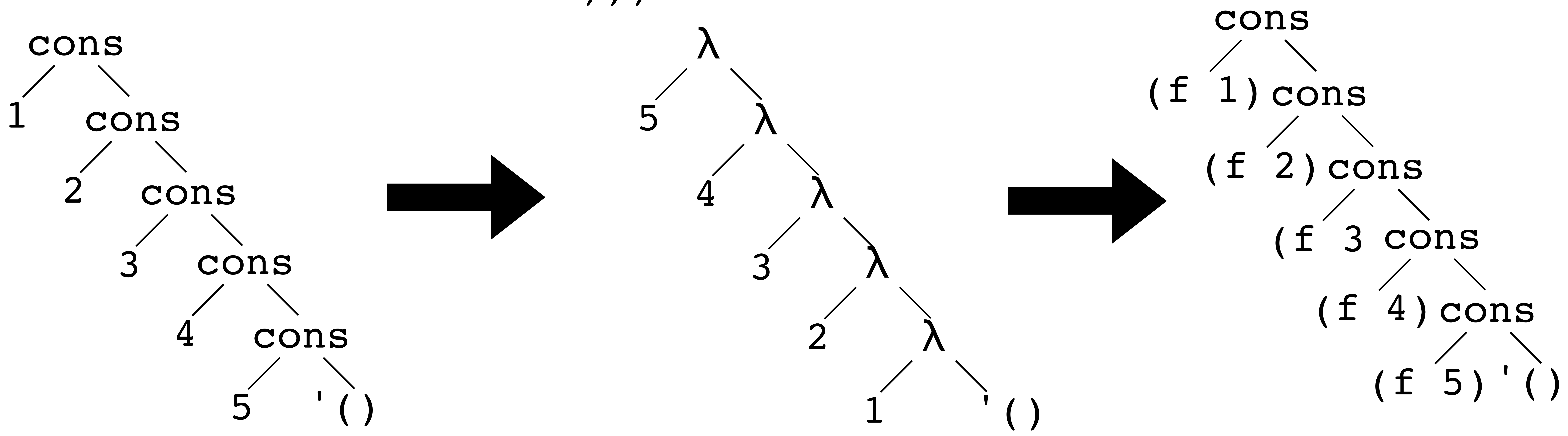
`initial-val: list of α`

`lst: list of α`

map as fold left

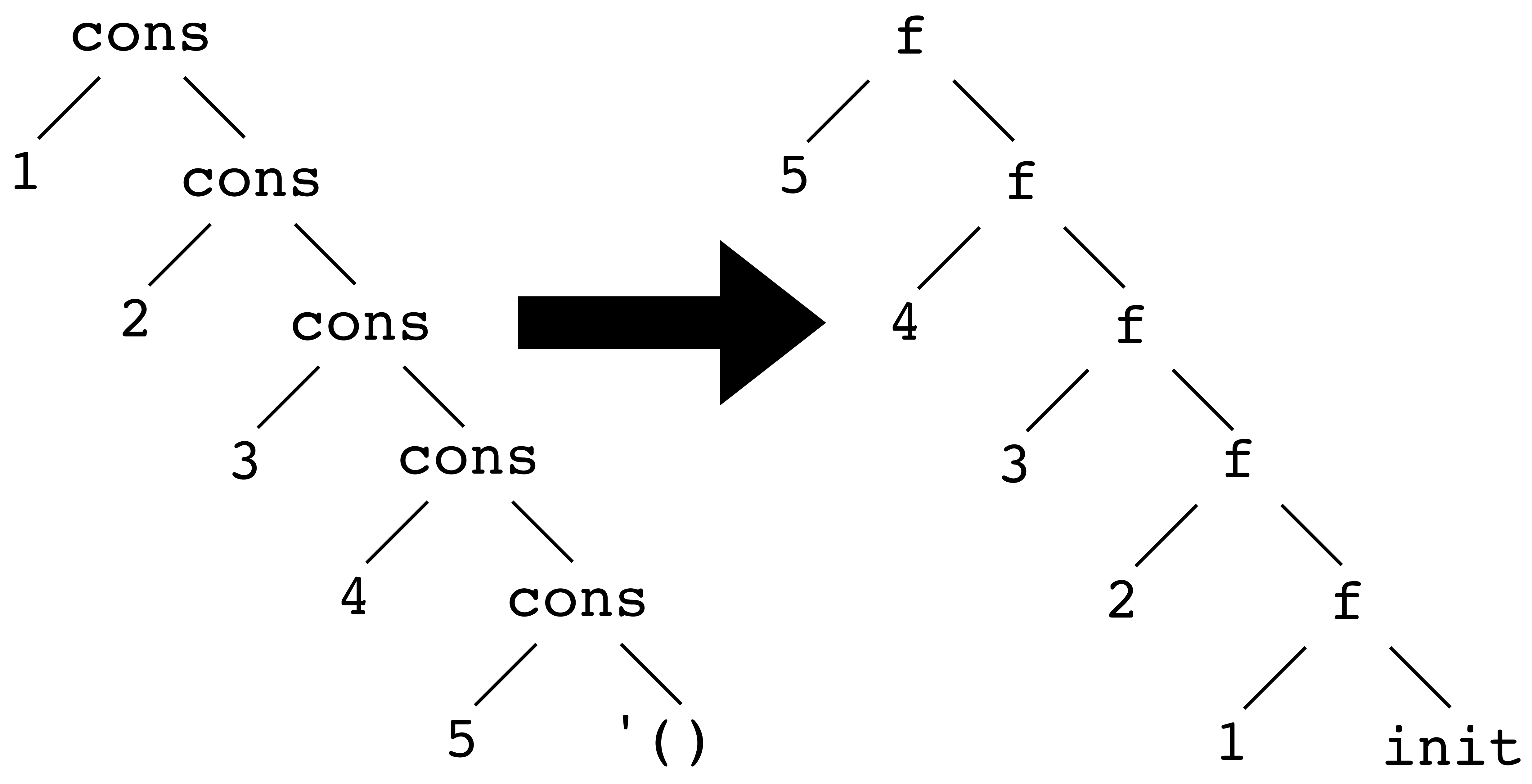
`(foldl combine initial-val lst)`

```
(define (map f lst)
  (reverse (foldl (λ (head acc)
                    (cons (f head) acc))
                  empty
                  lst)))
```

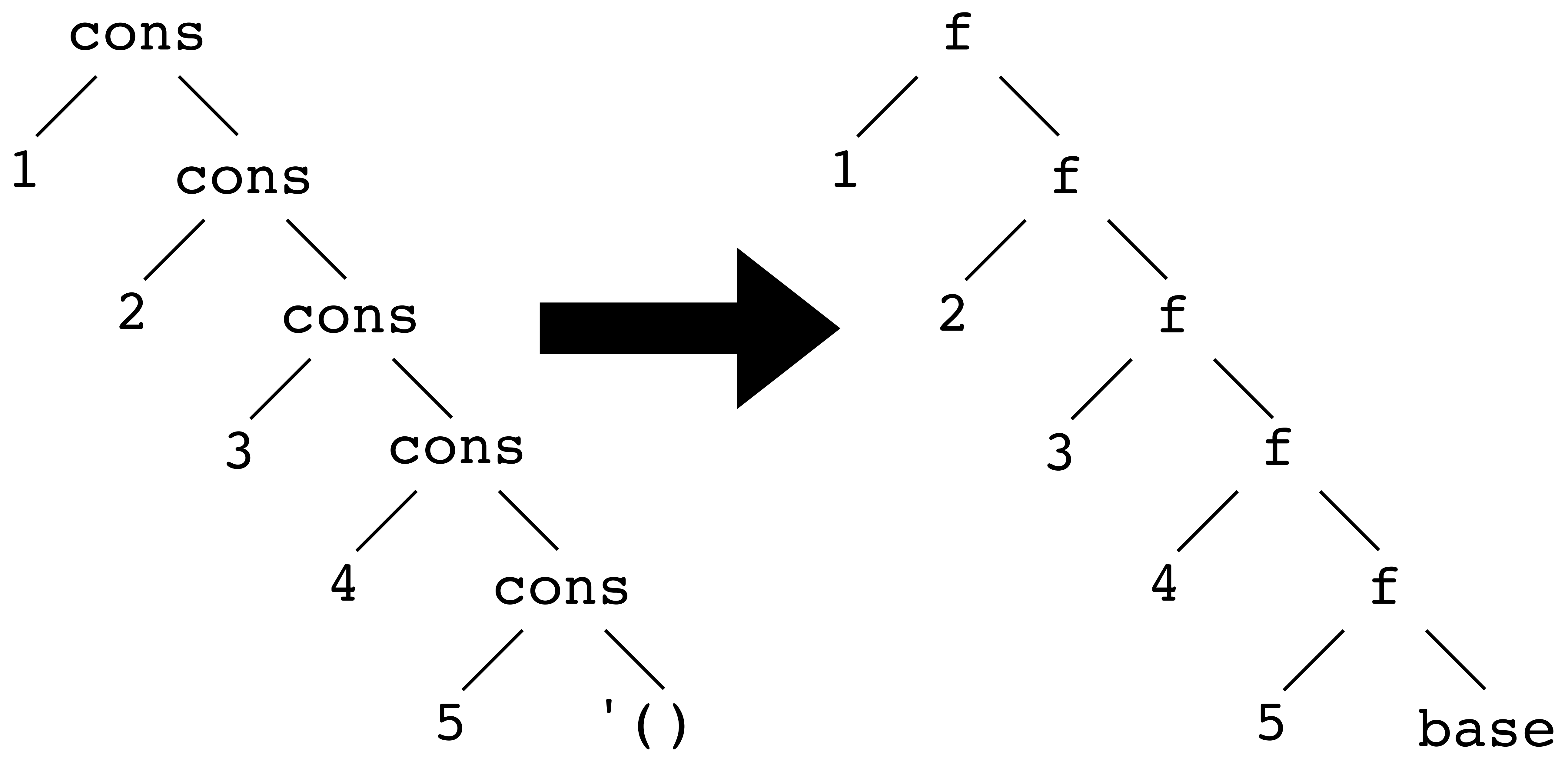


Both folds

`foldl`

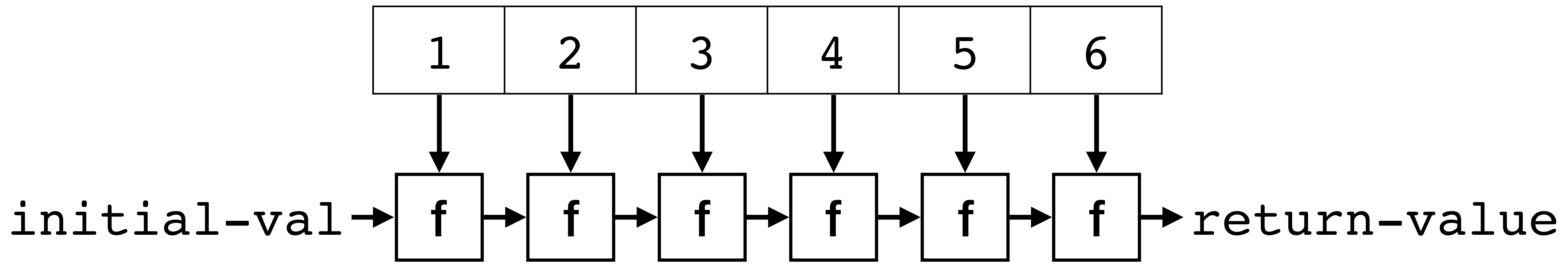


`foldr`

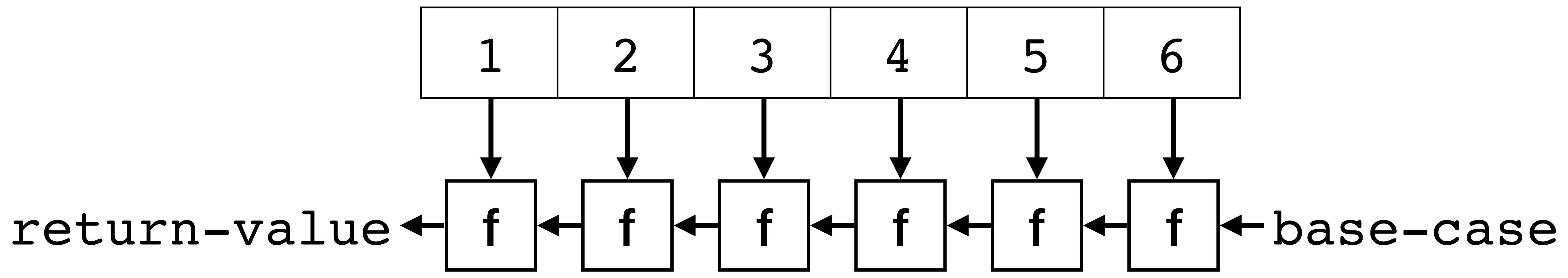


foldl vs. foldr

`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



Which is tail-recursive?

```
(define (foldr combine base lst)
  (cond [(empty? lst) base]
        [else (combine (first lst)
                       (foldr combine base (rest lst)))]))

(define (foldl combine initial-val lst)
  (cond [(empty? lst) initial-val]
        [else (foldl combine
                     (combine (first lst) initial-val)
                     (rest lst))]))
```

A. `foldl`

B. `foldr`

C. Both `foldl` and `foldr`

D. Neither `foldl` nor `foldr`