

# Programming Abstractions

**Week 3-2: Folds**

**Stephen Checkoway**

# Lots of similarities between functions

**(sum lst)**

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

# Lots of similarities between functions

**(length lst)**

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

# Lots of similarities between functions

**(map proc lst)**

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

# Lots of similarities between functions

**(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)))]))
```

Let's rewrite this one 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

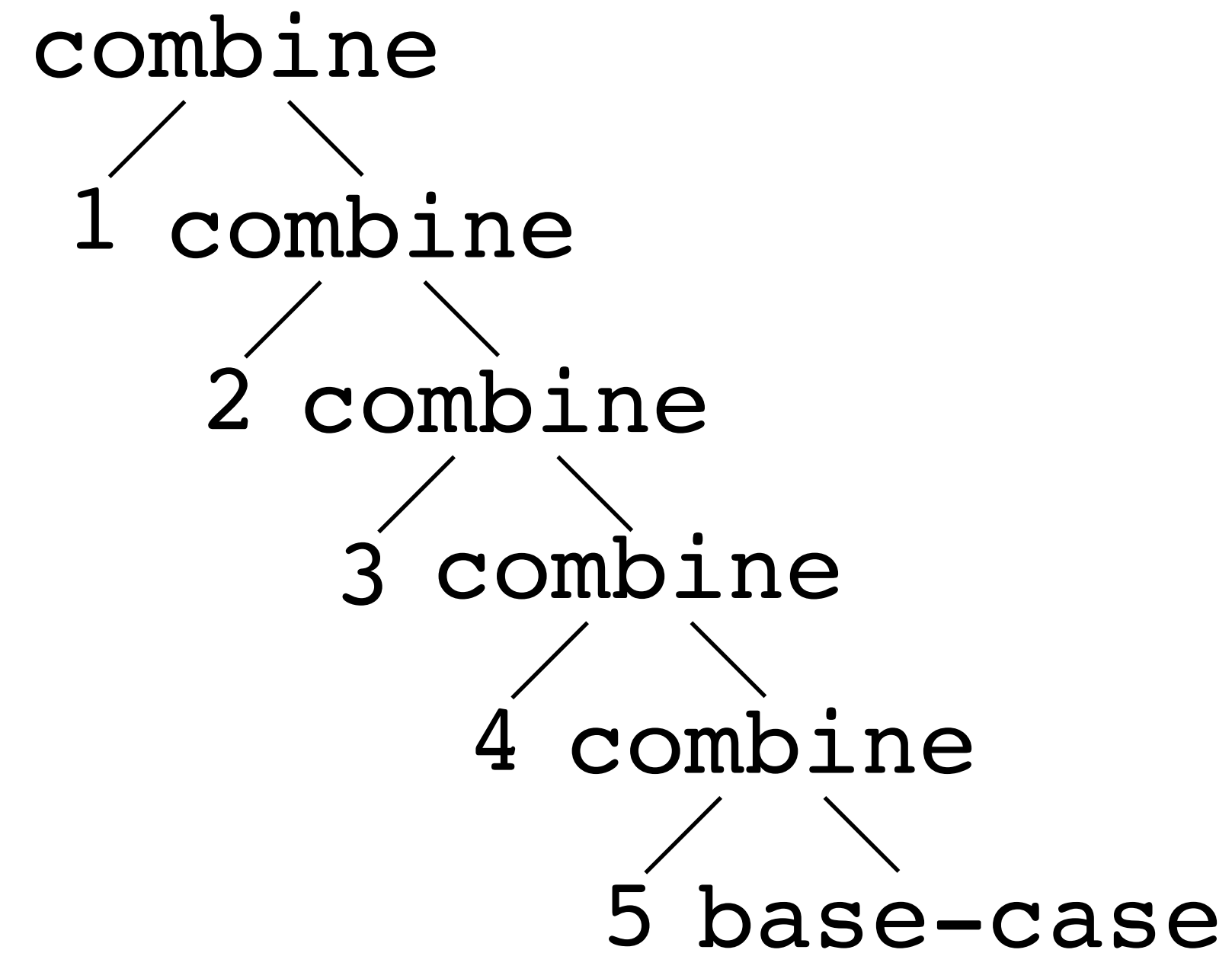
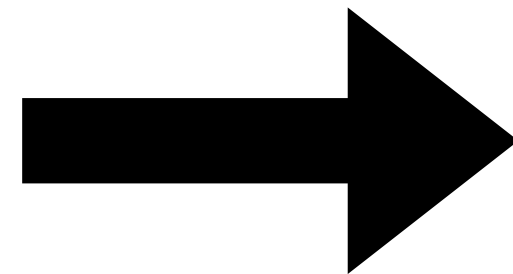
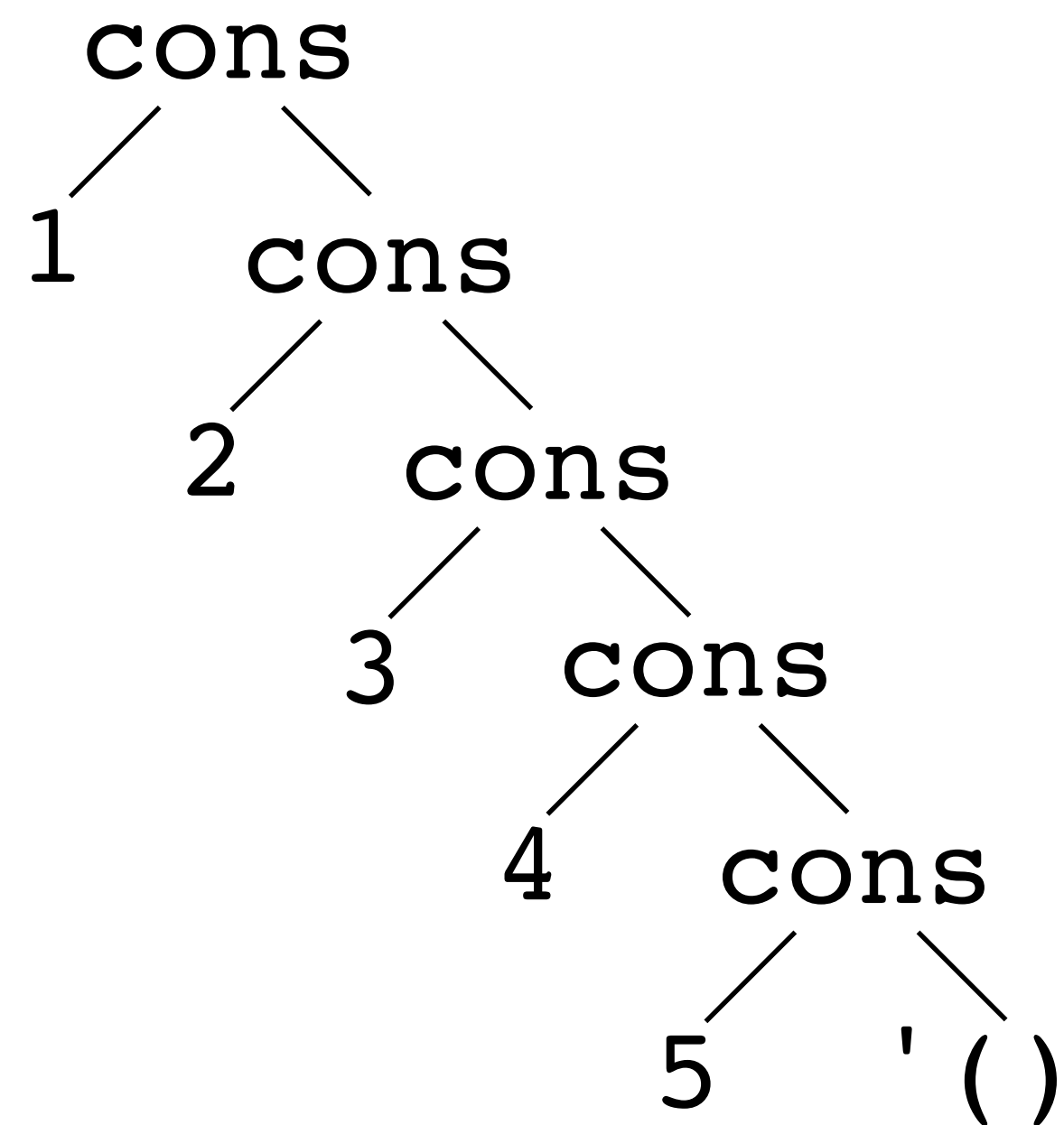
Basic structure is the same (rewriting slightly)

```
(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))

# Abstraction: fold right

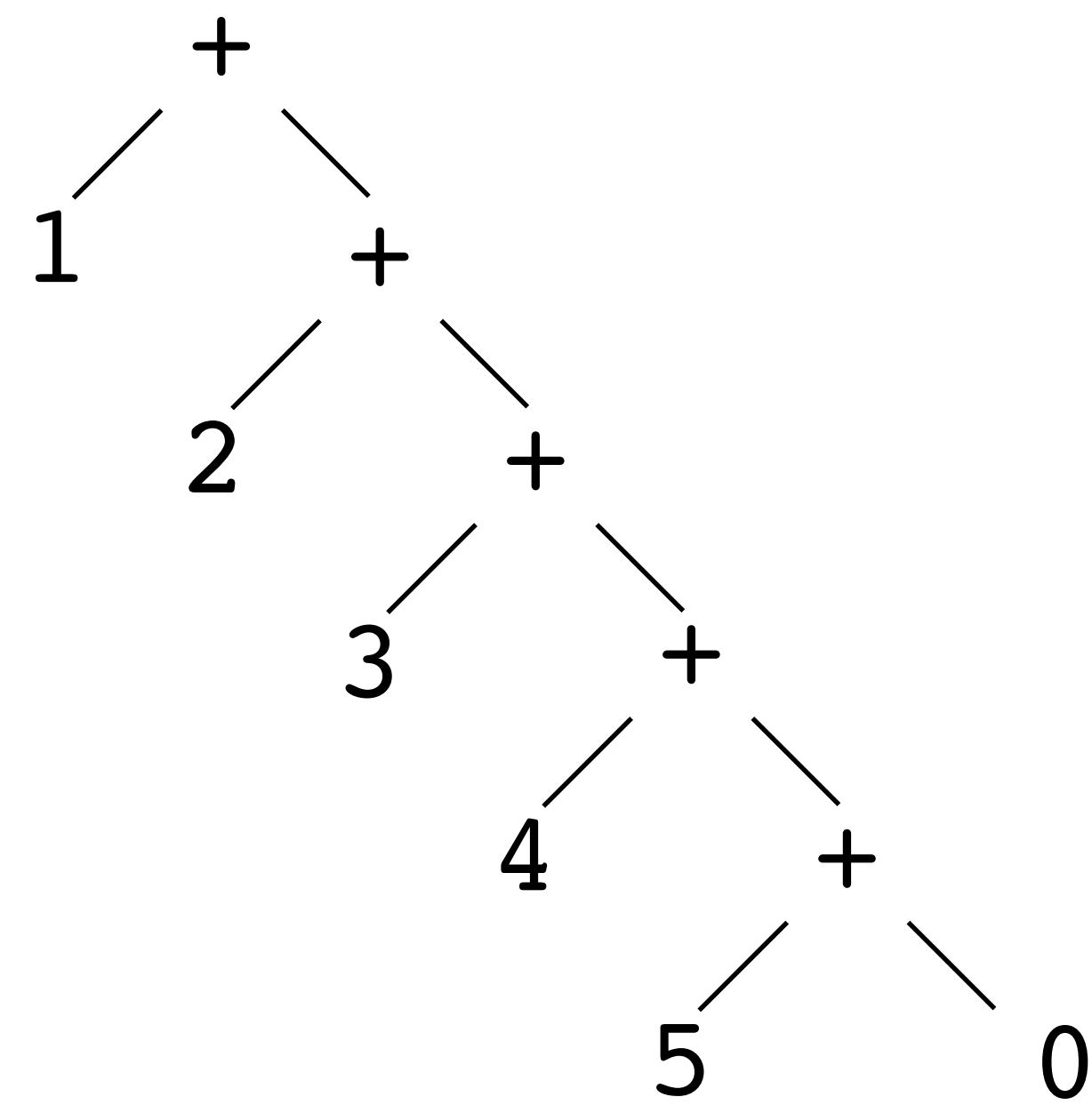
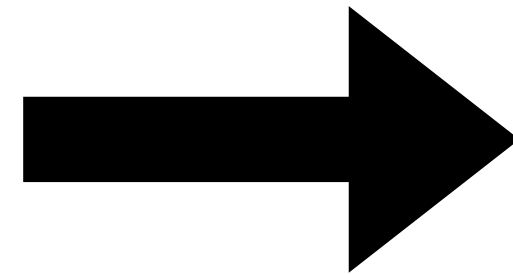
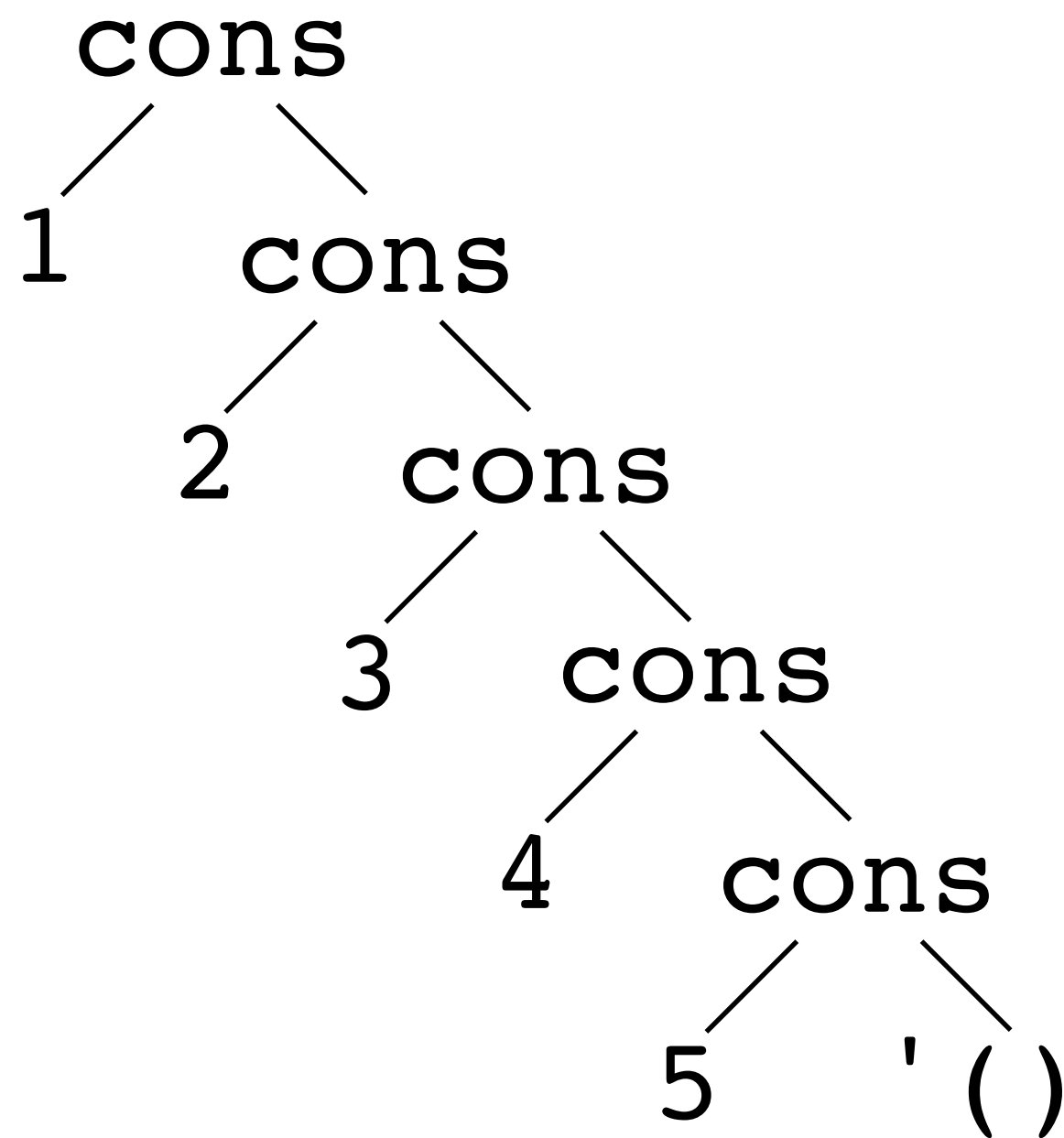
(`foldr combine base-case 1st`)



# sum as a fold right

**(foldr combine base-case lst)**

```
(define (sum lst)
  (foldr + 0 lst))
```

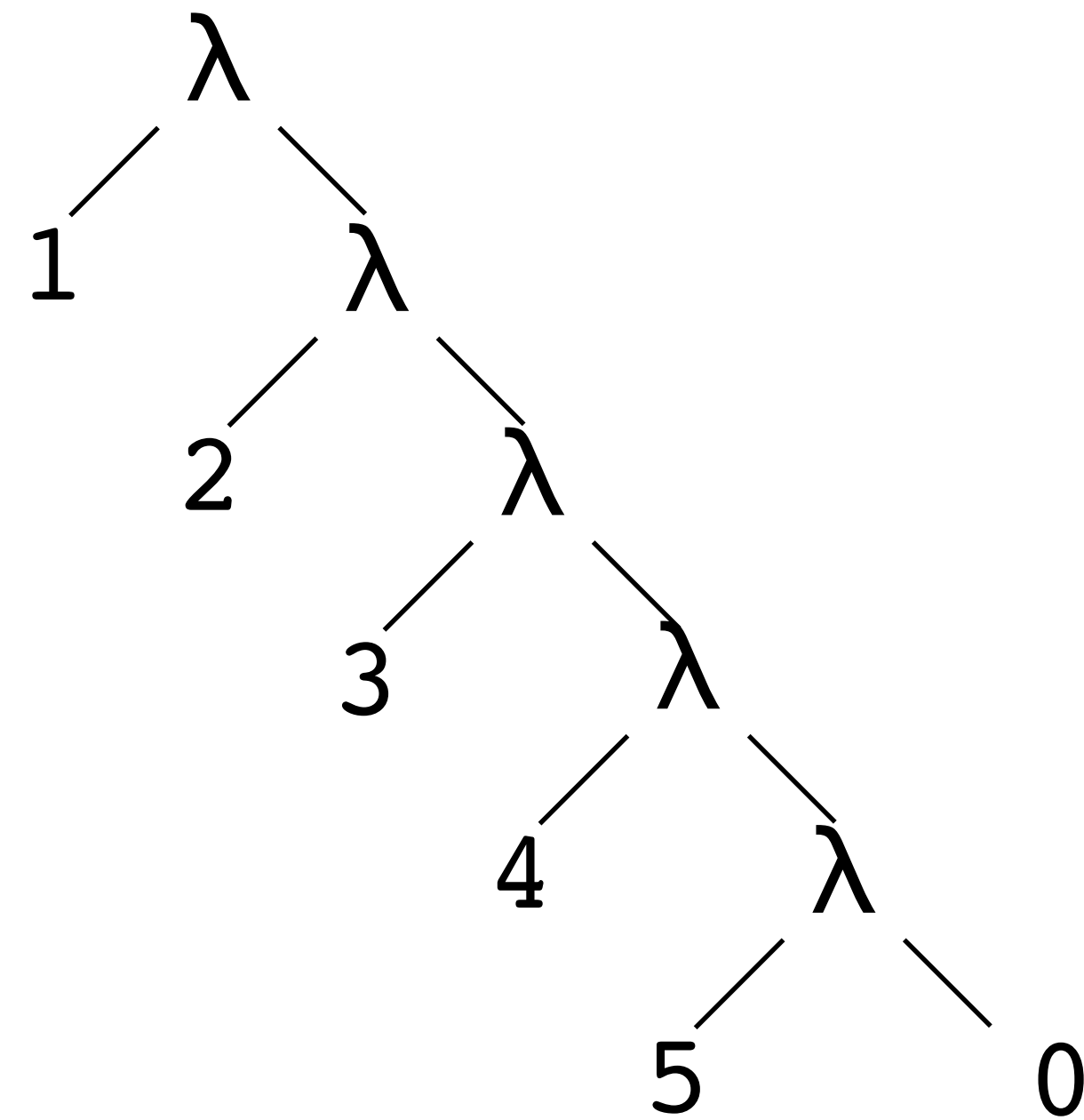
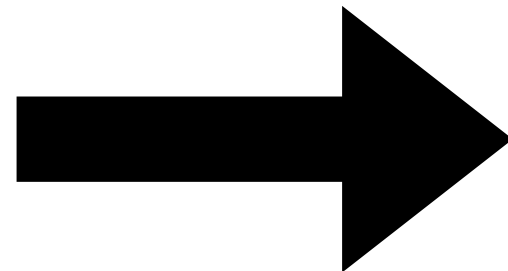
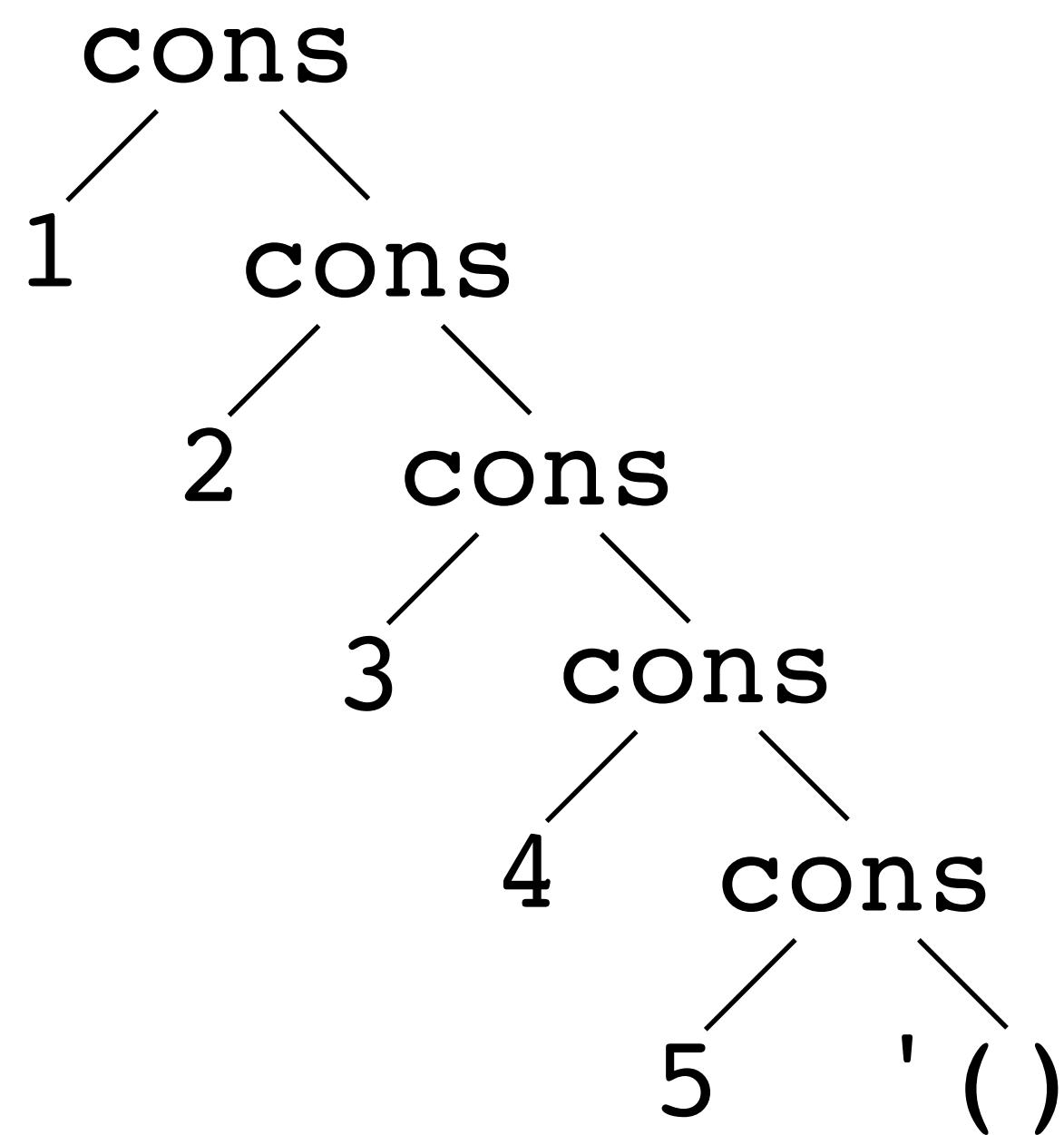




# length as a fold right

**(foldr combine base-case lst)**

```
(define (length lst)
  (foldr (λ (head result) (+ 1 result)) 0 lst))
```



# map and remove\* as fold right

(foldr combine base-case lst)

```
(define (map proc lst)
  (foldr (λ (head result)
          (cons (proc head) result))
        empty
        lst))
```

```
(define (remove* x lst)
  (foldr (λ (head result)
          (if (equal? x head)
              result
              (cons head result)))
        empty
        lst))
```

Consider the procedure

```
(define (foo lst)
  (foldr (λ (head result)
           (+ (* head head) result))
        0
        lst))
```

What is the result of `(foo '(1 0 2))`?

A. `'(1 0 2)`

B. `'(5 4 4)`

C. 5

D. 1

E. None of the above

Consider the procedure

```
(define (bar x lst)
  (foldr (λ (head result)
           (if (equal? head x) #t result))
        #f
        lst))
```

What is the result of `(bar 25 '(1 4 9 16 25 36 49))`?

A. `'(#f #f #f #f #t #f #f)`

B. `'(#f #f #f #f #t #t #t)`

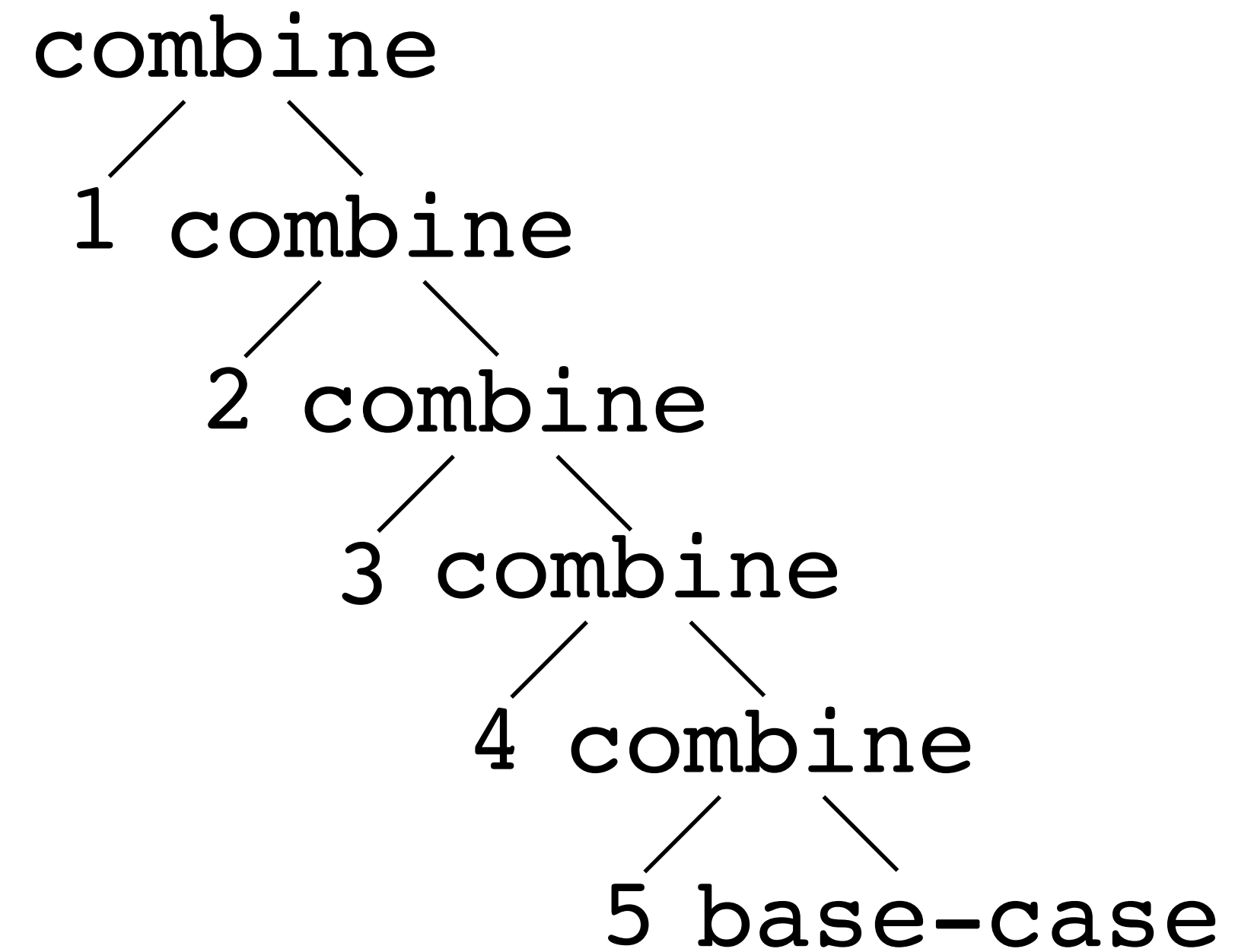
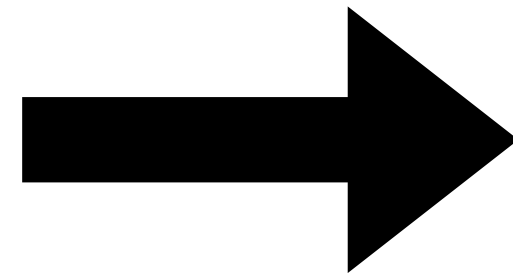
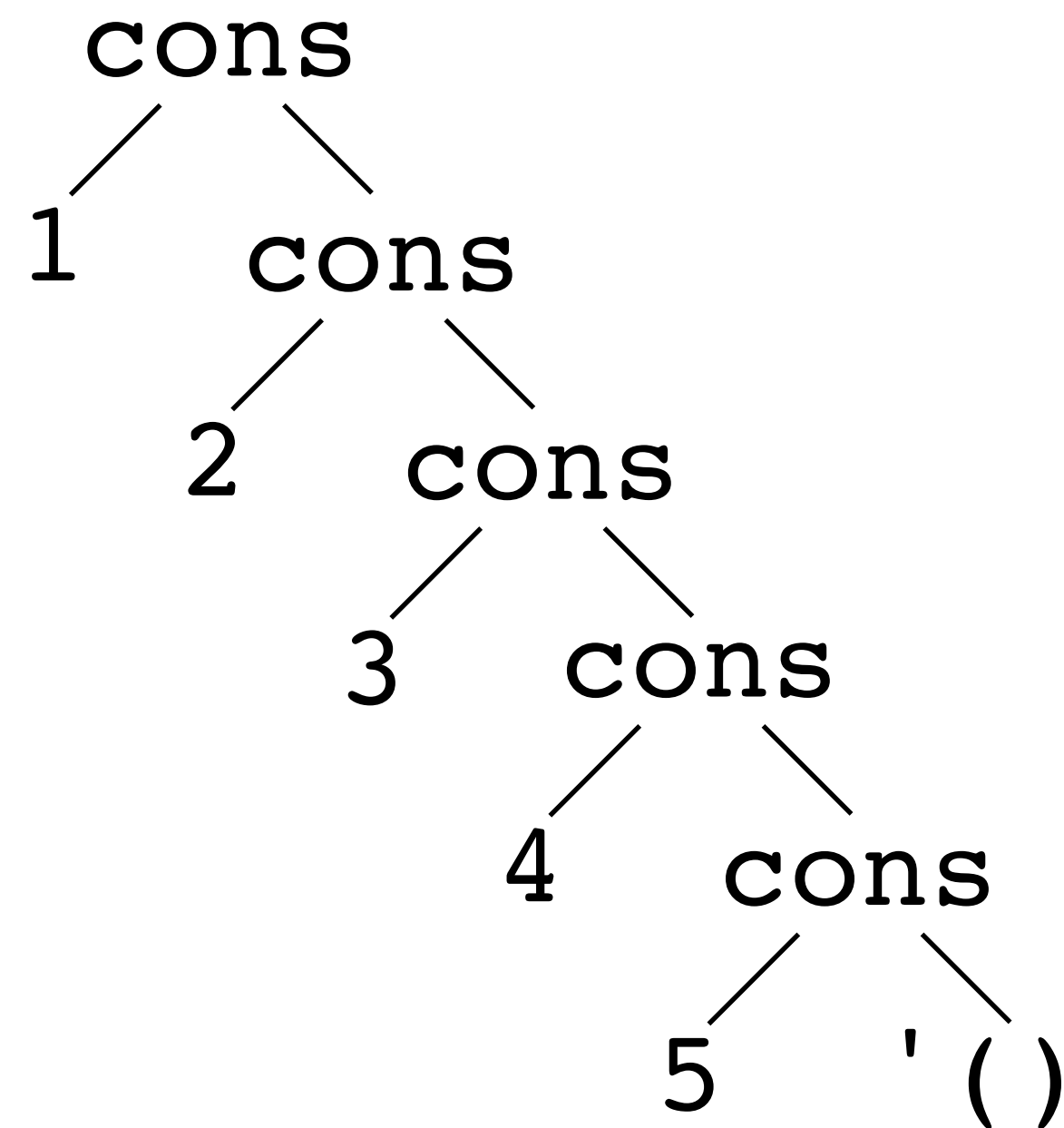
C. `#f`

D. `#t`

E. None of the above

# Let's write foldr

(foldr combine base-case 1st)



# Accumulation-passing style similarities

```
(define (product lst)
  (define (product-a lst acc)
    (cond [(empty? lst) acc]
          [else (product-a (rest lst)
                           (* (first lst) acc))]))
  (product-a lst 1))
```

# Accumulation-passing style similarities

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

# Accumulation-passing style similarities

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



# Some similarities

Basic structure is the same (rewriting slightly)

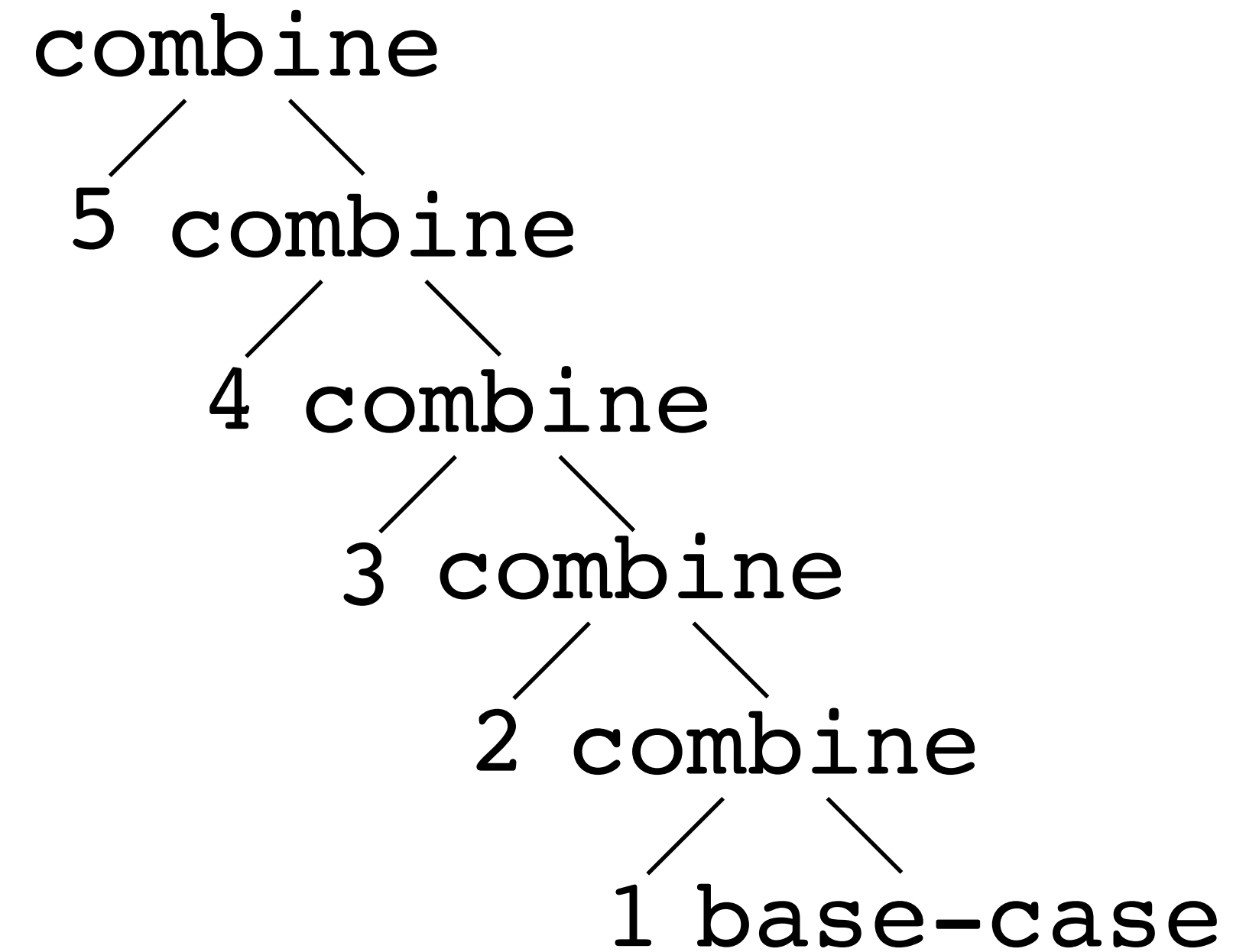
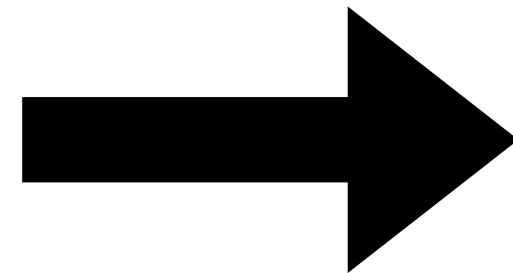
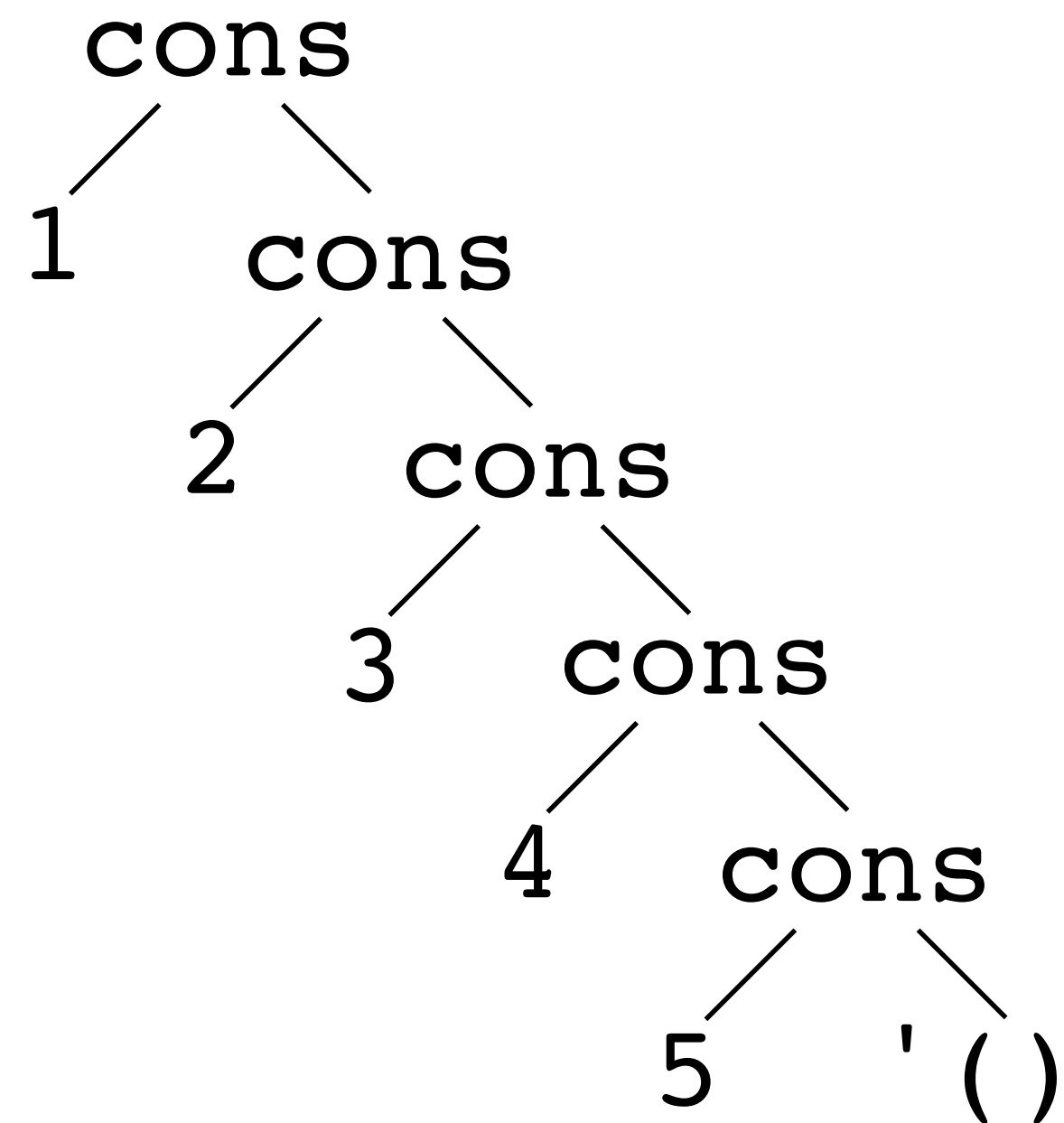
```
(define (fun ... lst)
  (define (fun-a lst acc)
    (cond [(empty? lst) acc]
          [else
           (fun-a (rest lst)
                   (combine (first lst) acc))]))
  (fun-a lst base-case))
```

Function	base-case	(combine head acc)
product	1	(* head acc)
reverse	empty	(cons head acc)
map	empty	(cons (proc head) acc)

We must reverse the result

# Abstraction foldl

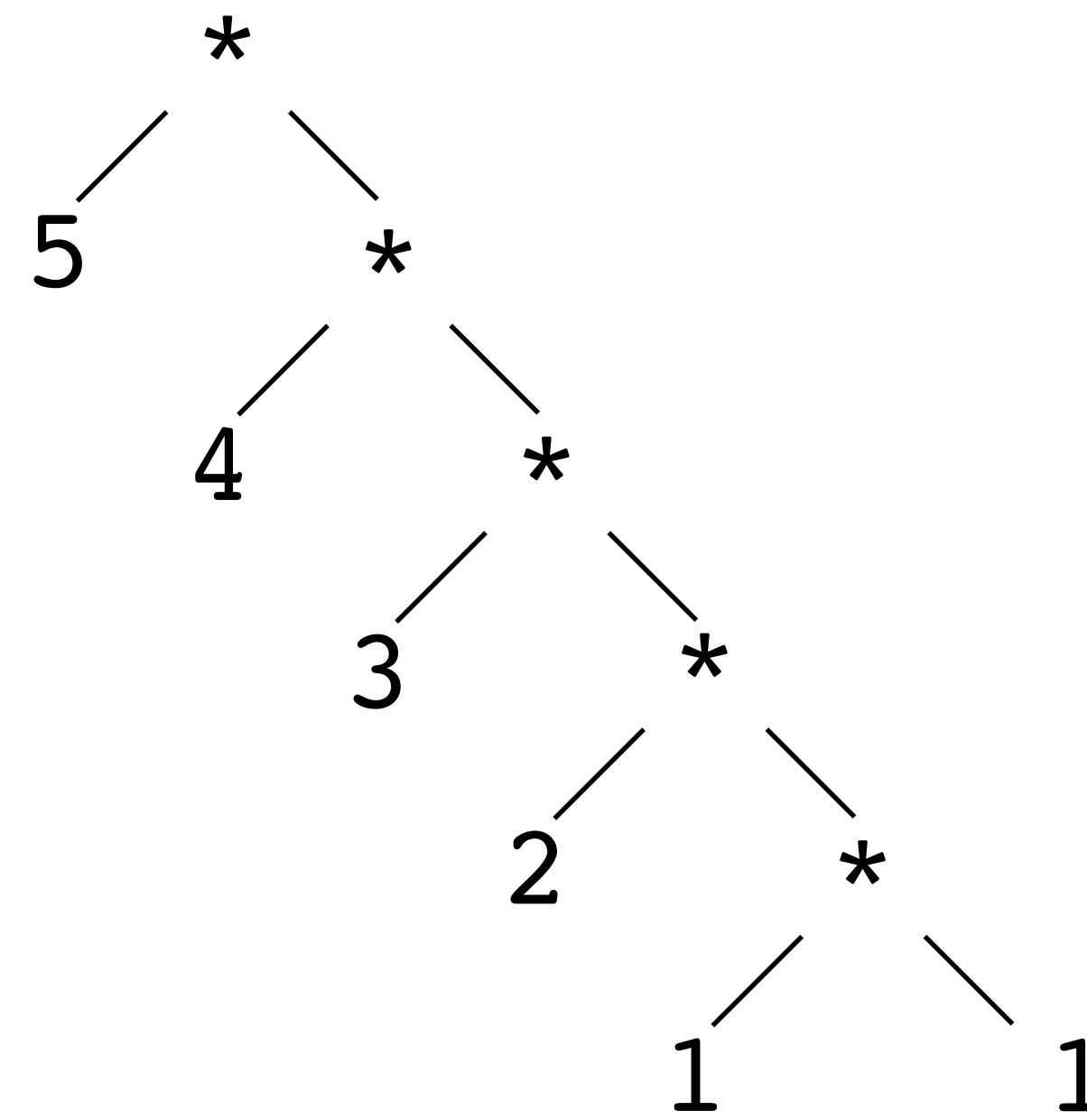
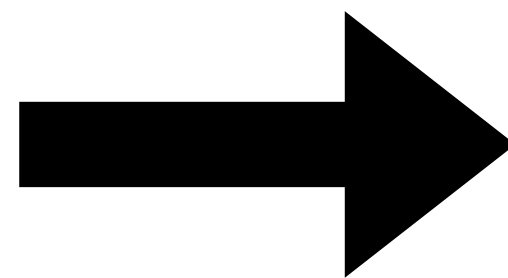
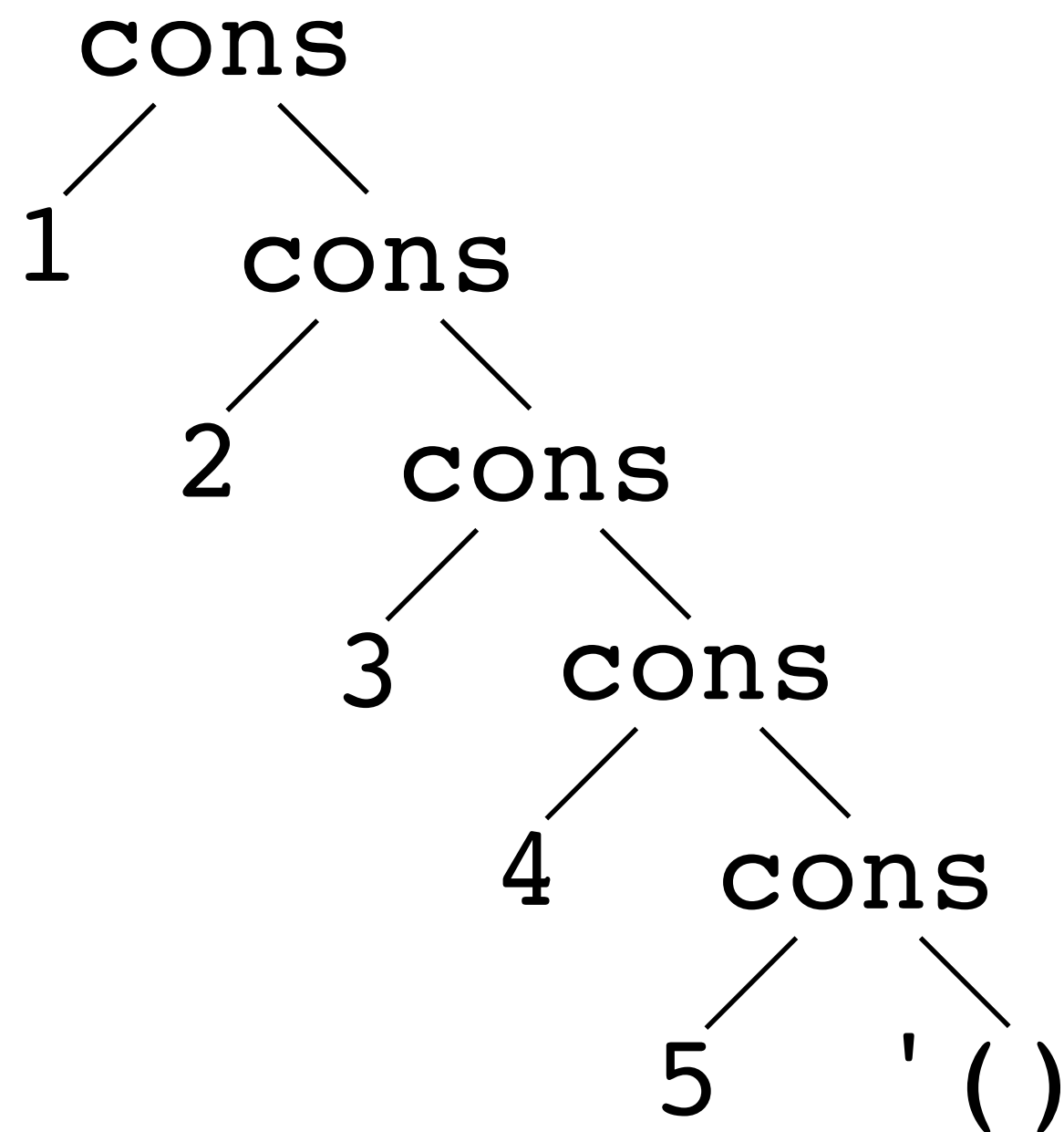
(foldl combine base-case 1st)



# product as fold left

**(foldl combine base-case lst)**

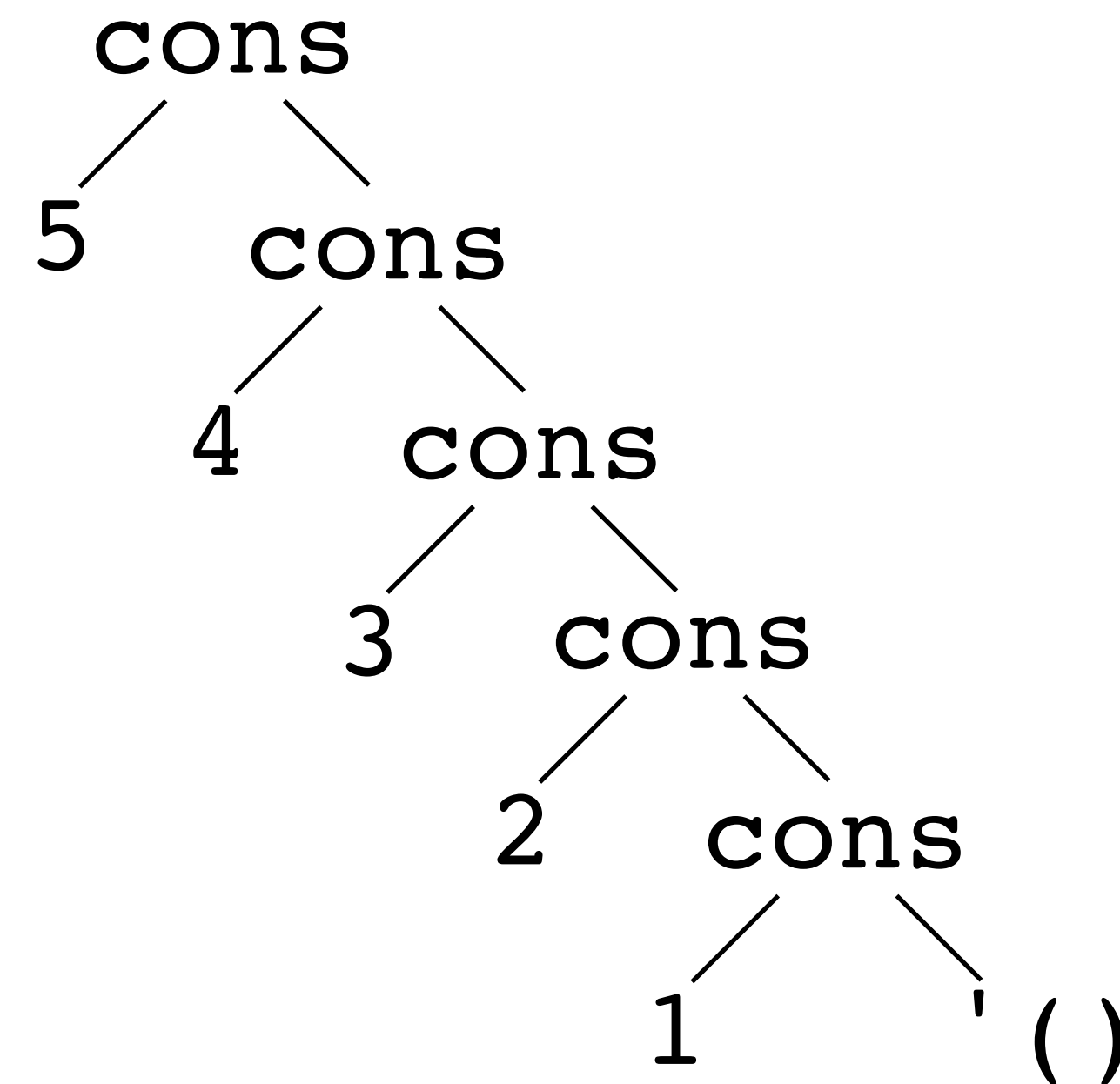
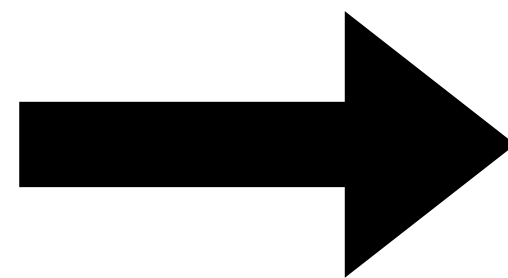
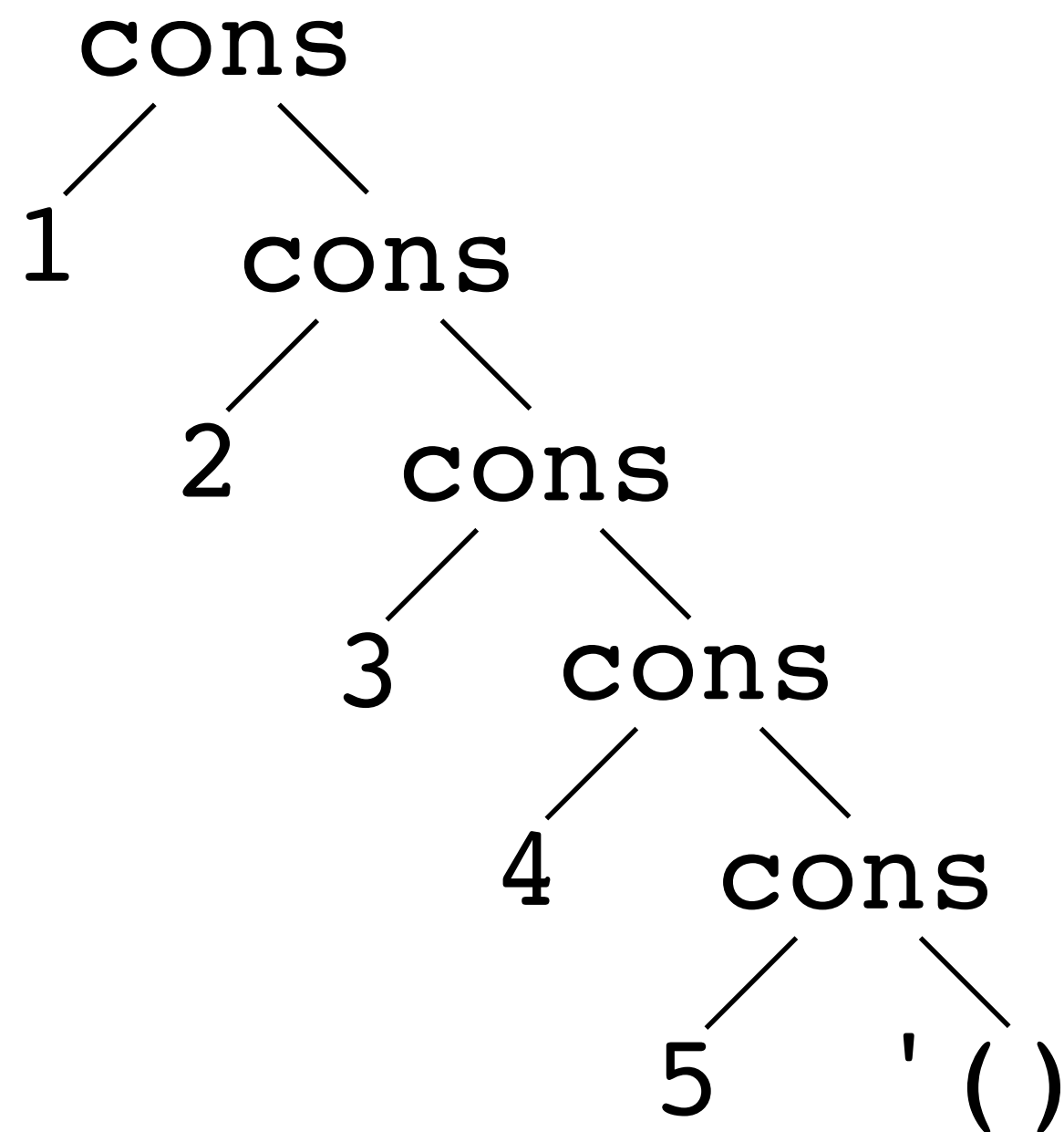
```
(define (product lst)
  (foldl * 1 lst))
```



# reverse as fold left

**(foldl combine base-case lst)**

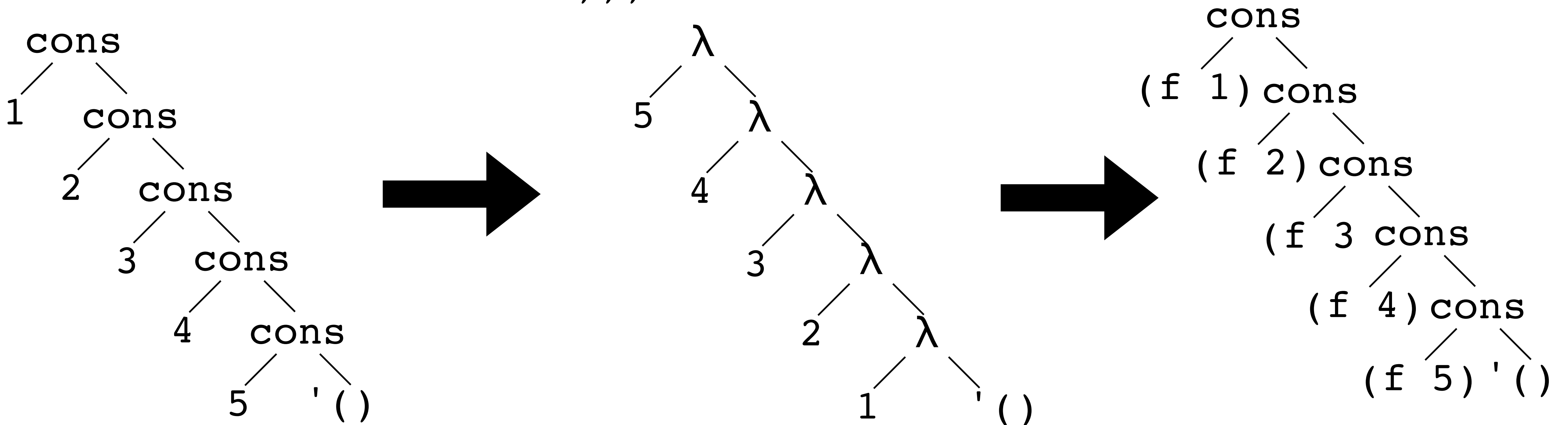
```
(define (reverse lst)
  (foldl cons empty lst))
```



# reverse as fold left

**(foldl combine base-case lst)**

```
(define (map f lst)
  (reverse (foldl ( $\lambda$  (head acc)
                        (cons (f head) acc))
                  empty
                  lst)))
```



# Let's write `remove*` using `foldl`

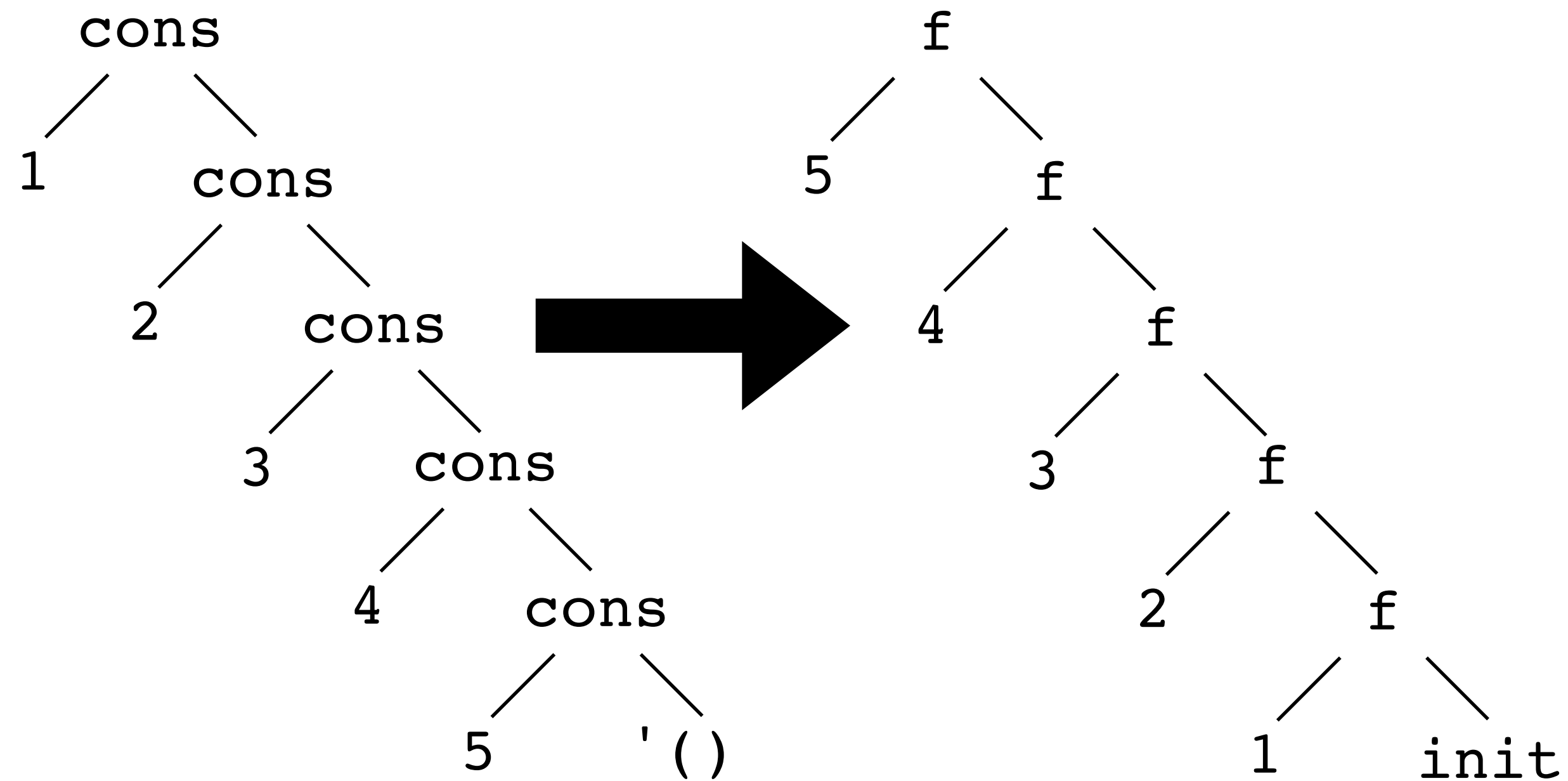
`(foldl combine base-case lst)`

`combine` has the form `( $\lambda$  (head acc) ...)`

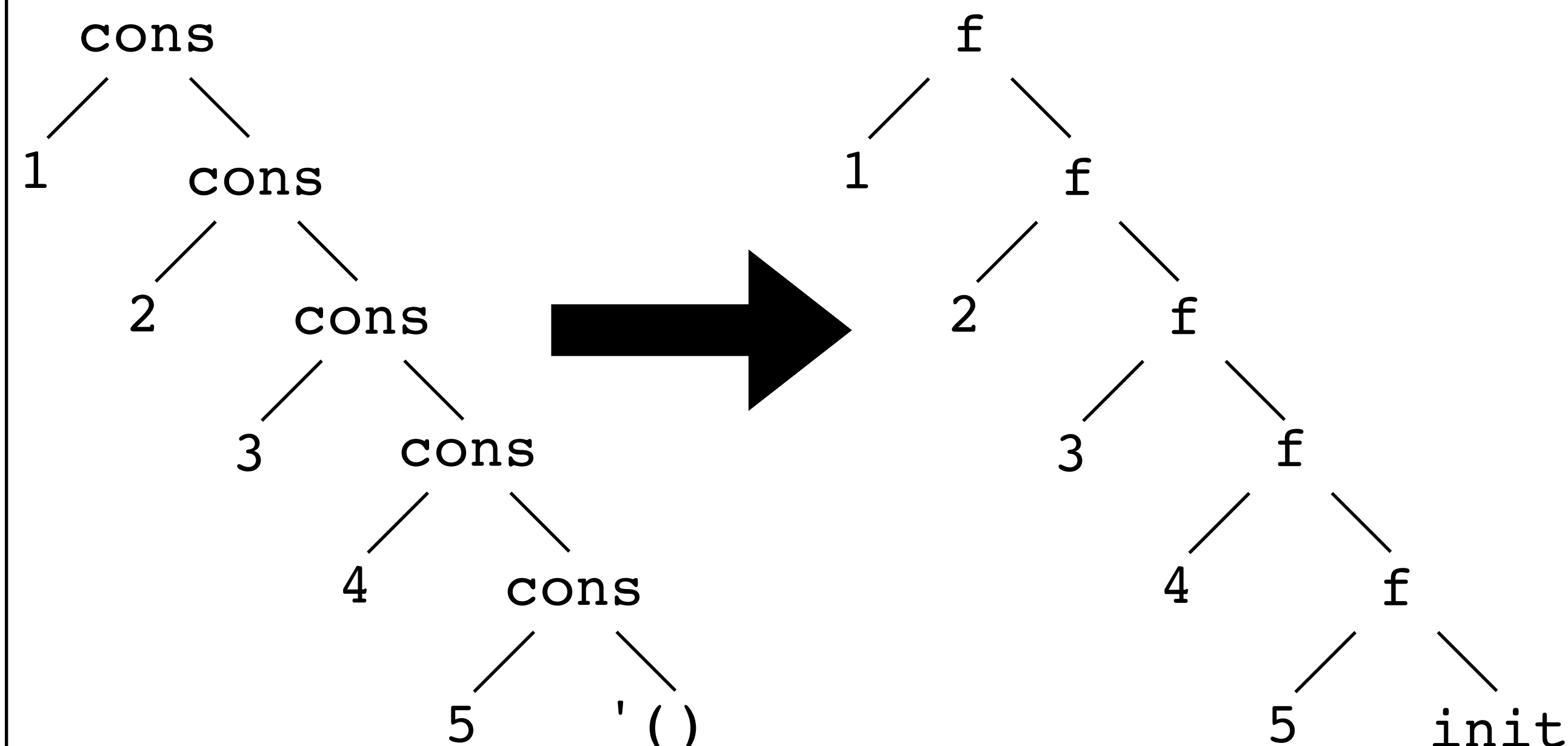
We'll need to reverse the result!

# Both folds

foldl

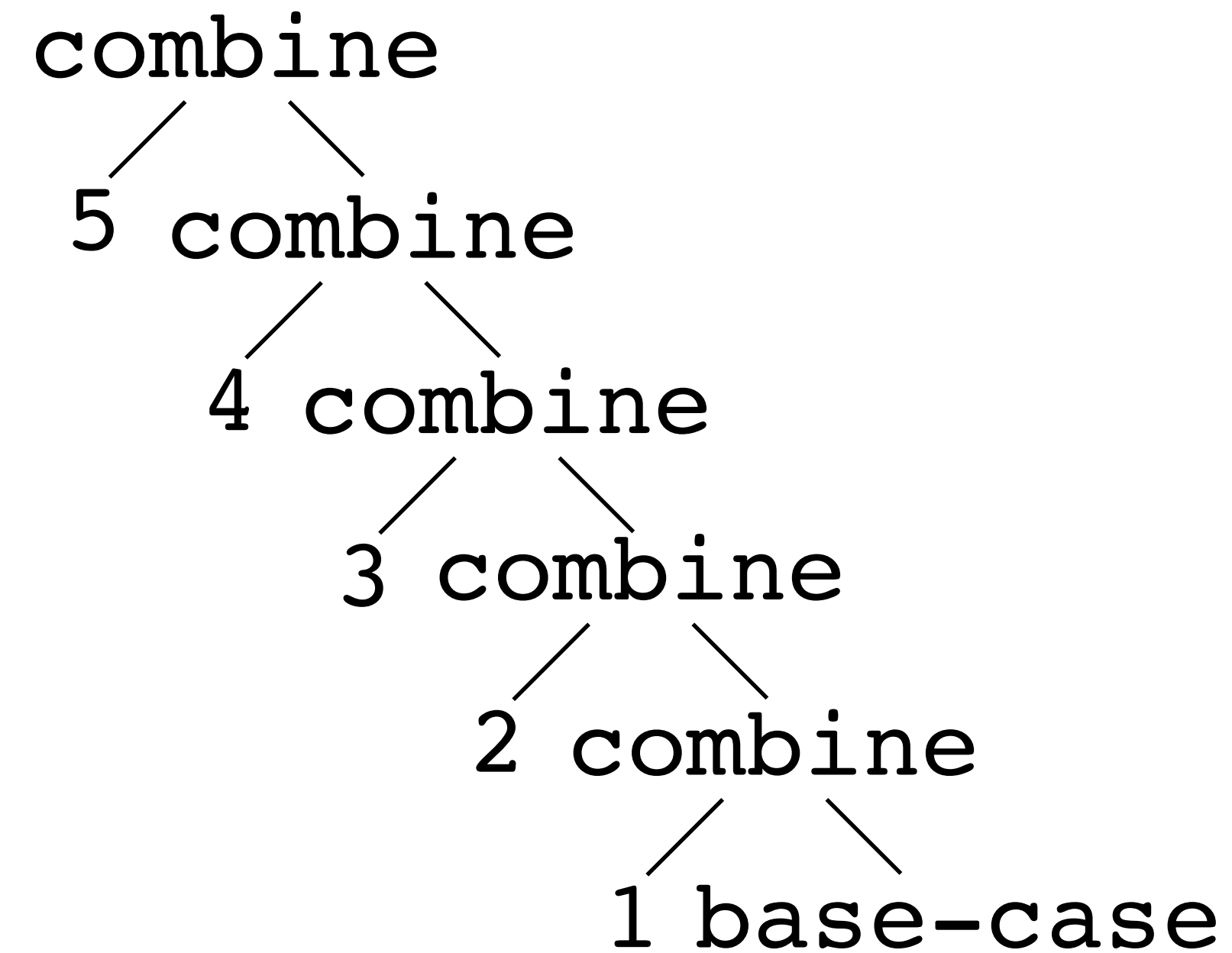
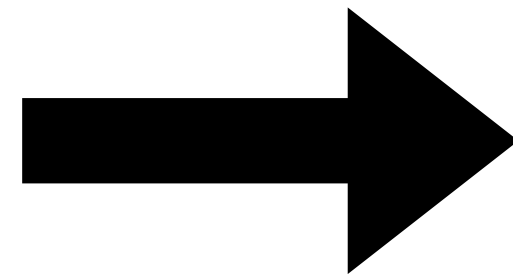
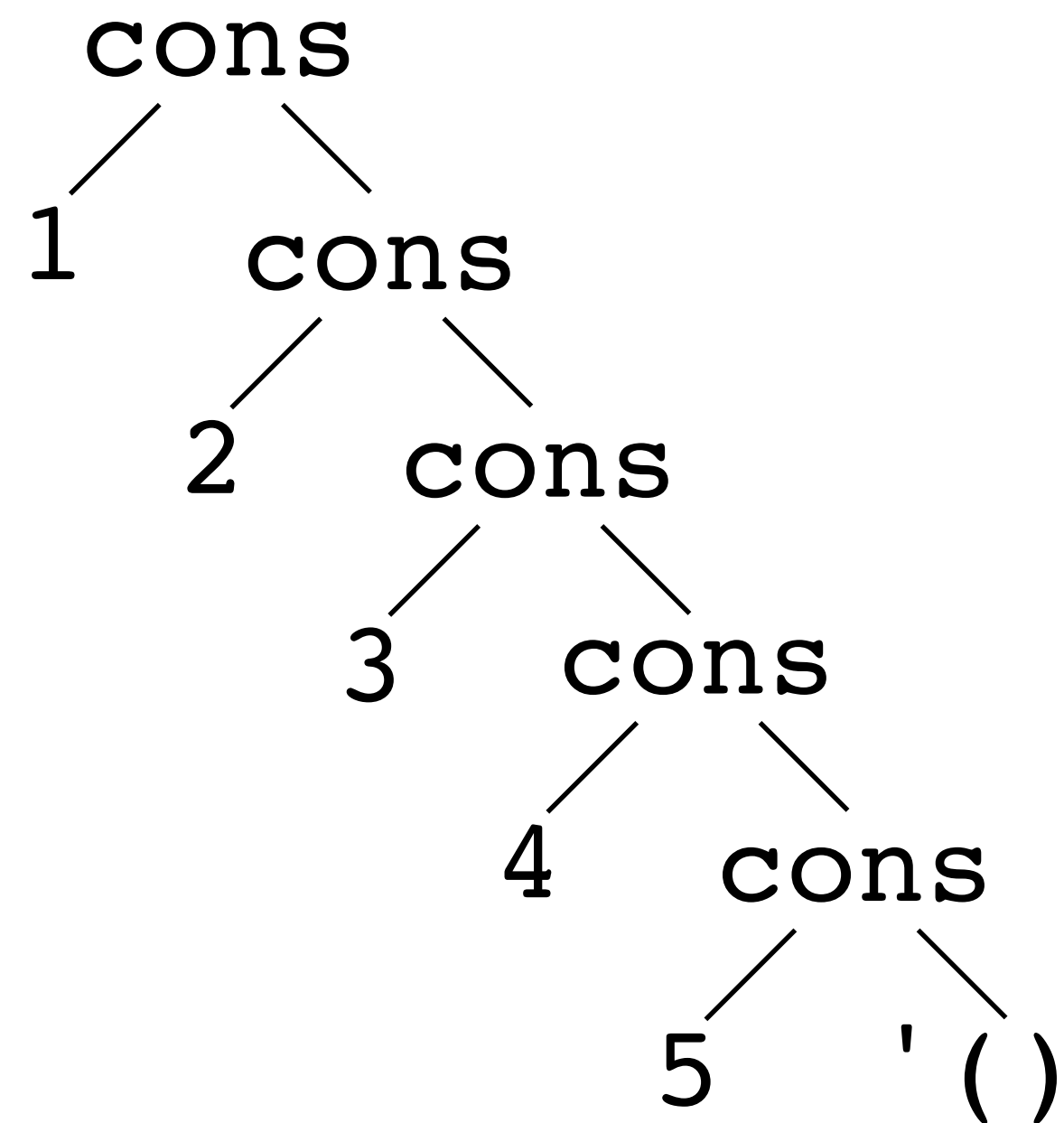


foldr



# Let's write foldl

(foldl combine base-case 1st)





Which is tail-recursive?

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

```
(define (foldl f init lst)
  (cond [(empty? lst) init]
        [else (foldl f
                      (f (first lst) init)
                      (rest lst))]))
```

A. foldl

C. Both foldl and foldr

B. foldr

D. Neither foldl nor foldr