Programming Abstractions

Lecture 9: Fold right

Stephen Checkoway

Announcements

Homework 1 due on Friday

Three free late days

Office hours on Friday 13:30–14:30

Ask questions on Piazza!

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))

```
(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

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

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

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

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

```
(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`

Abstraction: fold right

`(foldr combine base-case lst)`

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

`base-case`: β

`lst`: `list of` α

`foldr`: $(\alpha \times \beta \rightarrow \beta) \times \beta \times$ `(list of` α`)` $\rightarrow \beta$

Elements of `lst = (x1 x2 ... xn)` and `base-case` are combined by computing

z_n = `(combine` x_n `base-case)`

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

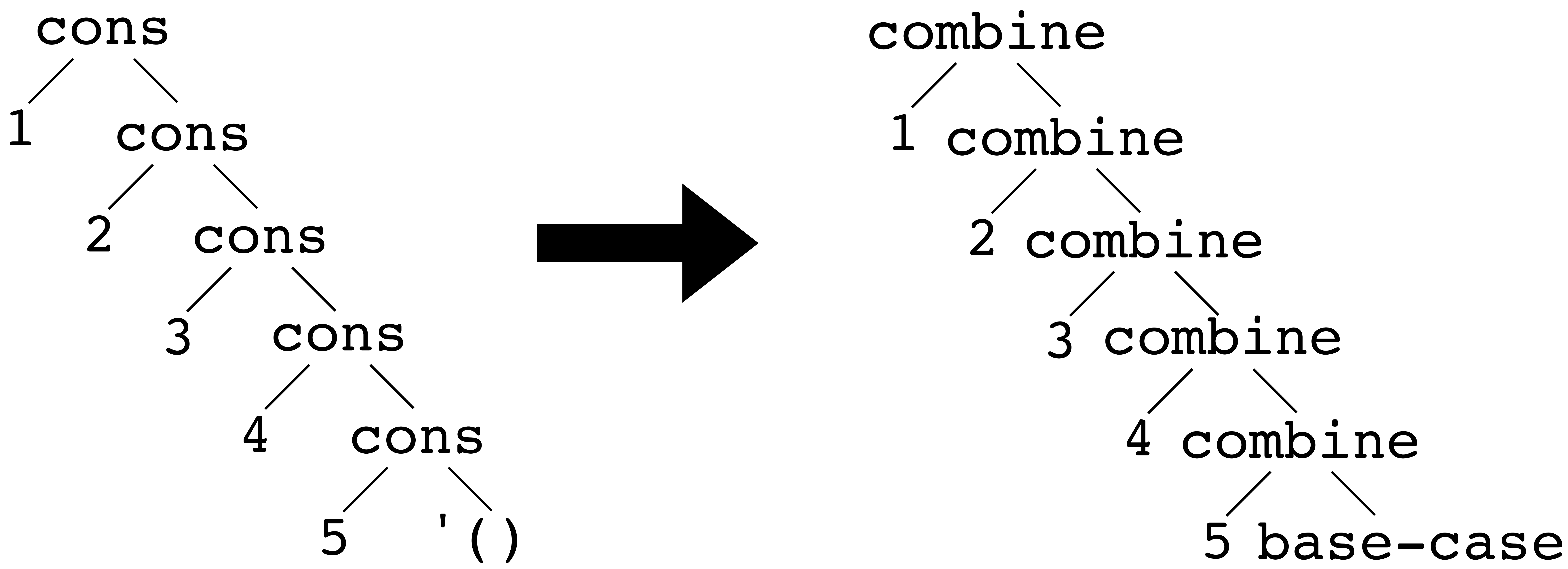
z_{n-2} = `(combine` x_{n-2} z_{n-1}`)`

$\vdots$

z_1 = `(combine` x_1 z_2`)`

Abstraction: fold right

`(foldr combine base-case lst)`

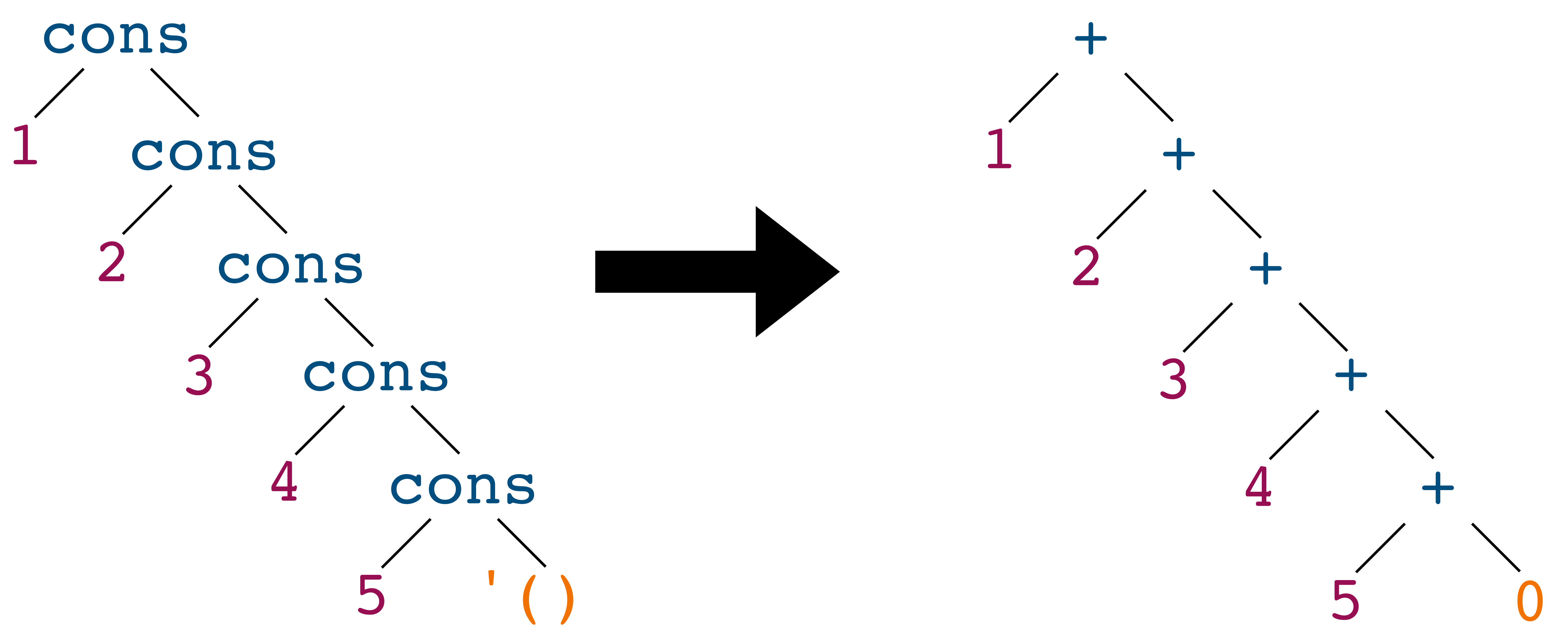


sum as a fold right

`(foldr combine base-case lst)`

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

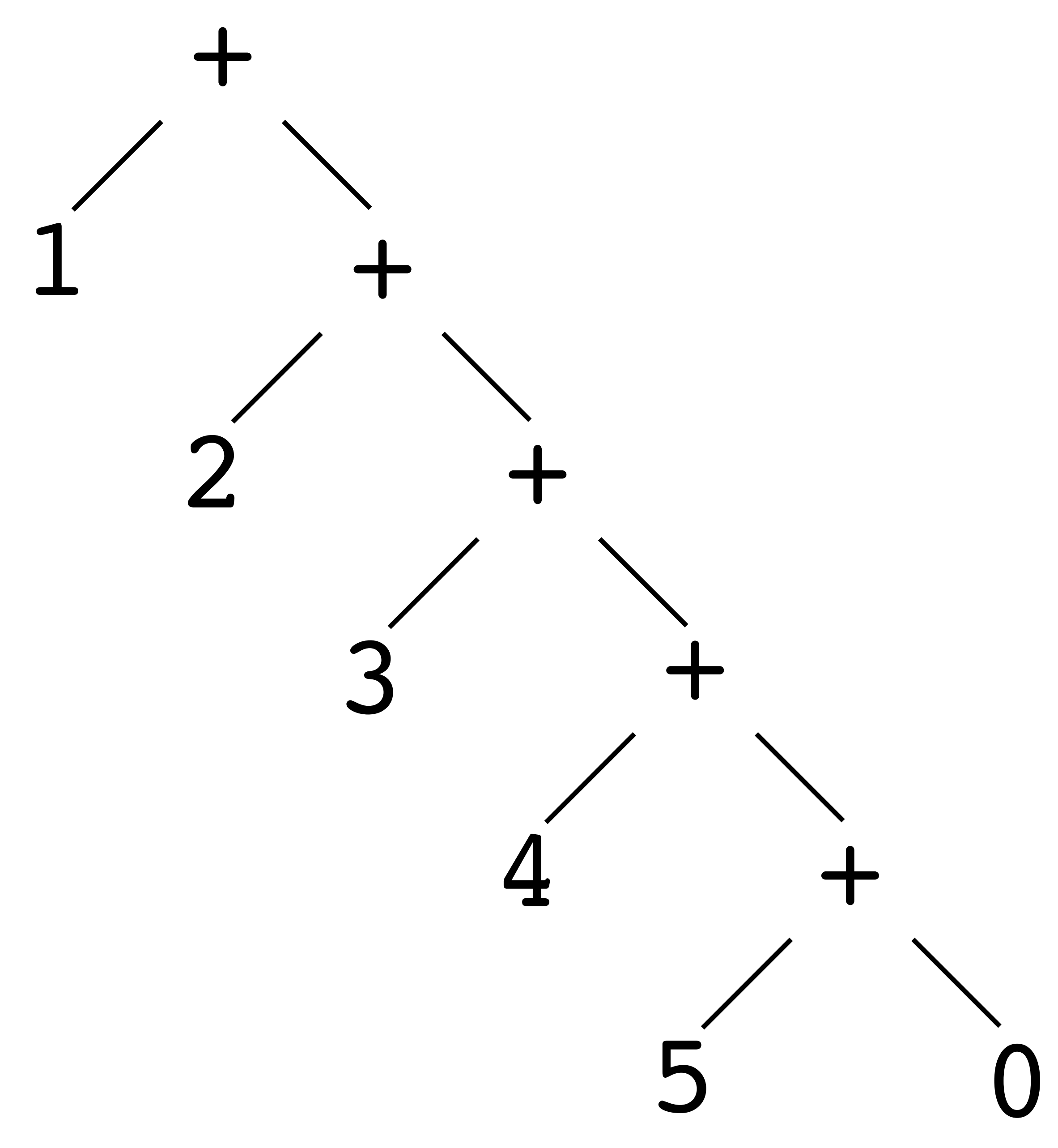
```
combine: number × number → number
base-case: number
lst: list of number
```



Print out the arguments

```
(foldr (λ (x acc)
         (let ([result (+ x acc)])
           (printf "(+ ~s ~s) => ~s~n" x acc result)
           result))
       0
       '(1 2 3 4 5))
```

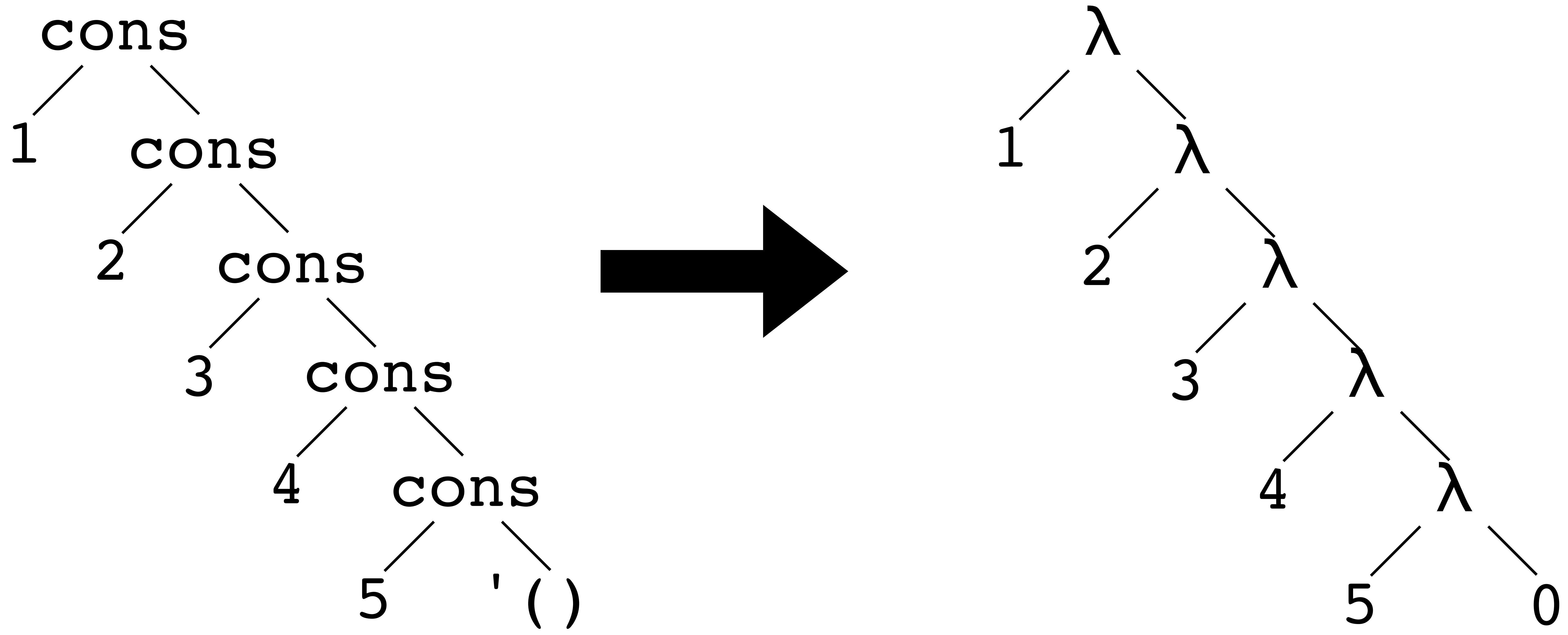
```
(+ 5 0) => 5
(+ 4 5) => 9
(+ 3 9) => 12
(+ 2 12) => 14
(+ 1 14) => 15
```



length as a fold right

`(foldr combine base-case lst)`

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



map as fold right

(foldr combine base-case lst)

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

proc: $\alpha \rightarrow \beta$

combine: $\alpha \times$ (list of β) $\rightarrow$ list of β

base-case: list of β

lst: list of α

map: $(\alpha \rightarrow \beta) \times$ (list of α) $\rightarrow$ list of β

`remove*` as fold right

`(foldr combine base-case lst)`

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

x: α

combine: $\alpha \times (\text{list of } \alpha) \rightarrow \text{list of } \alpha$

base-case: list of α

lst: list of α

remove*: $\alpha \times (\text{list of } \alpha) \rightarrow \text{list of } \alpha$

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

Example: a light switch state machine

Consider a light switch connected to a light

The light is in one of two states: on and off

- Represent this with symbols `'on` and `'off`

There are three actions we can take

- `'up`: move the switch to the up position; turns the light on
- `'down`: move the switch to the down position; turns the light off
- `'flip`: flip the position of the switch; changes the state of the light

If the light is initially `'off`, then after the sequence of actions
`'(up up down flip flip flip)`,
the light will be 'on

Implement the state machine

`(next-state action state)`

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

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

Write a `(next-state action state)` function that returns the next state of the light after the action is performed in the given state

Write a `(state-after actions)` that returns the state of the light assuming it's initially `'off` and the actions in the list `actions` are performed in order

- Use `foldr` and be careful about the order of operations

Let's write `foldr`

`(foldr combine base-case lst)`