Programming Abstractions

Week 3-2: Folds and Combinators

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 lst)`

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

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

Abstraction: fold right

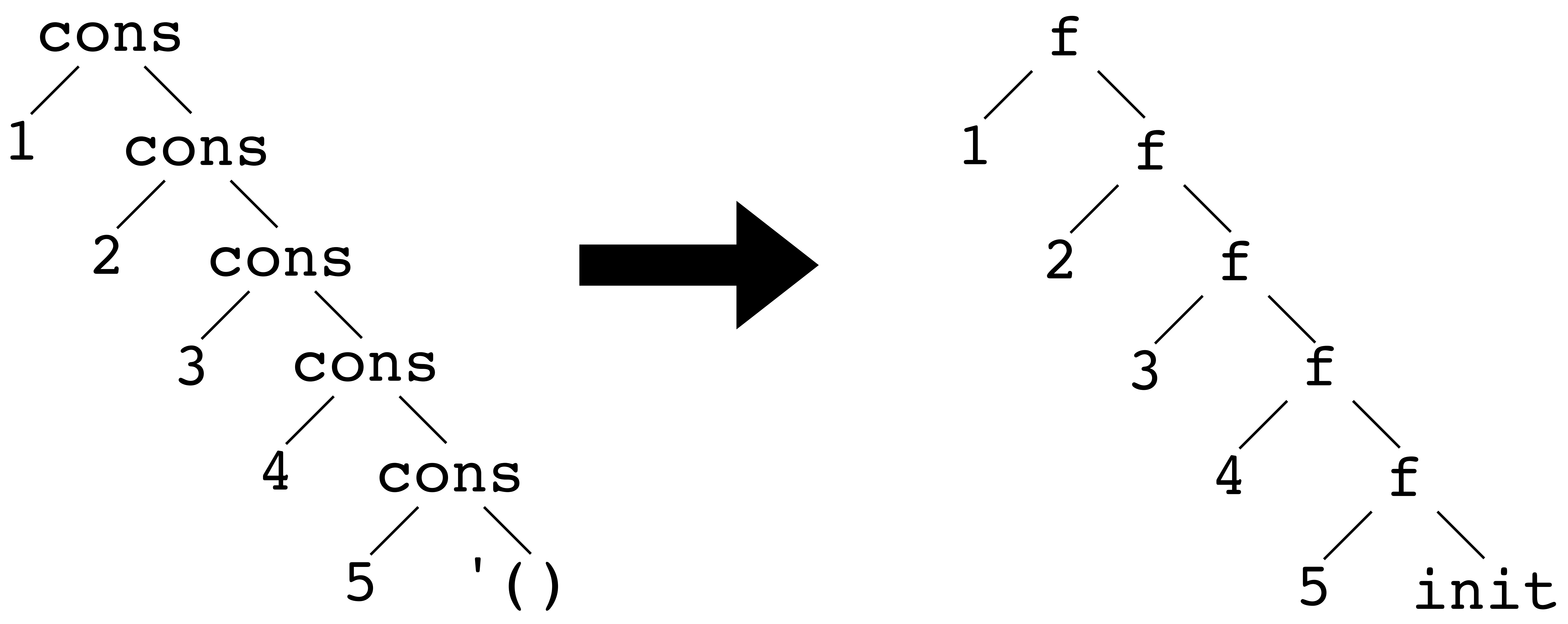
`(foldr combine base-case lst)`

```
(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))
```

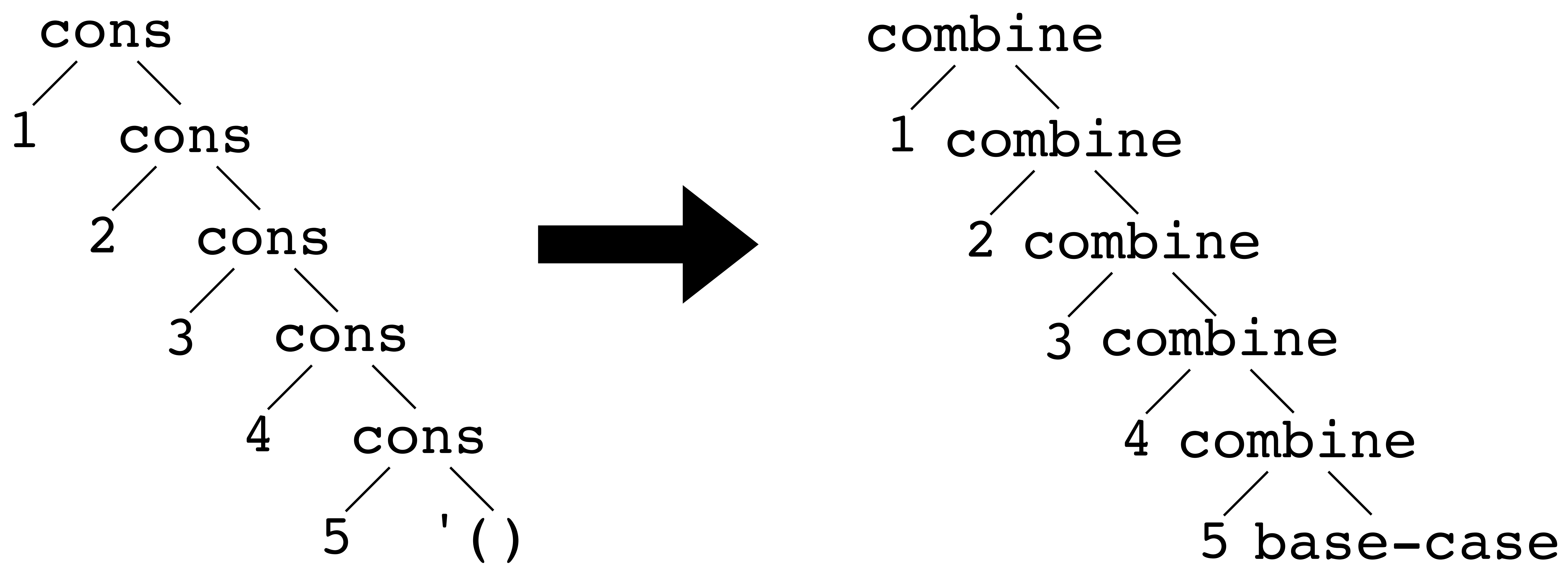
Visualizing foldl

```
(foldr f init '(1 2 3 4 5))
```



Let's write `foldr`

`(foldr combine base-case lst)`



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: fold left

`(foldl combine base-case lst)`

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

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

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

Let's write remove* using `foldl`

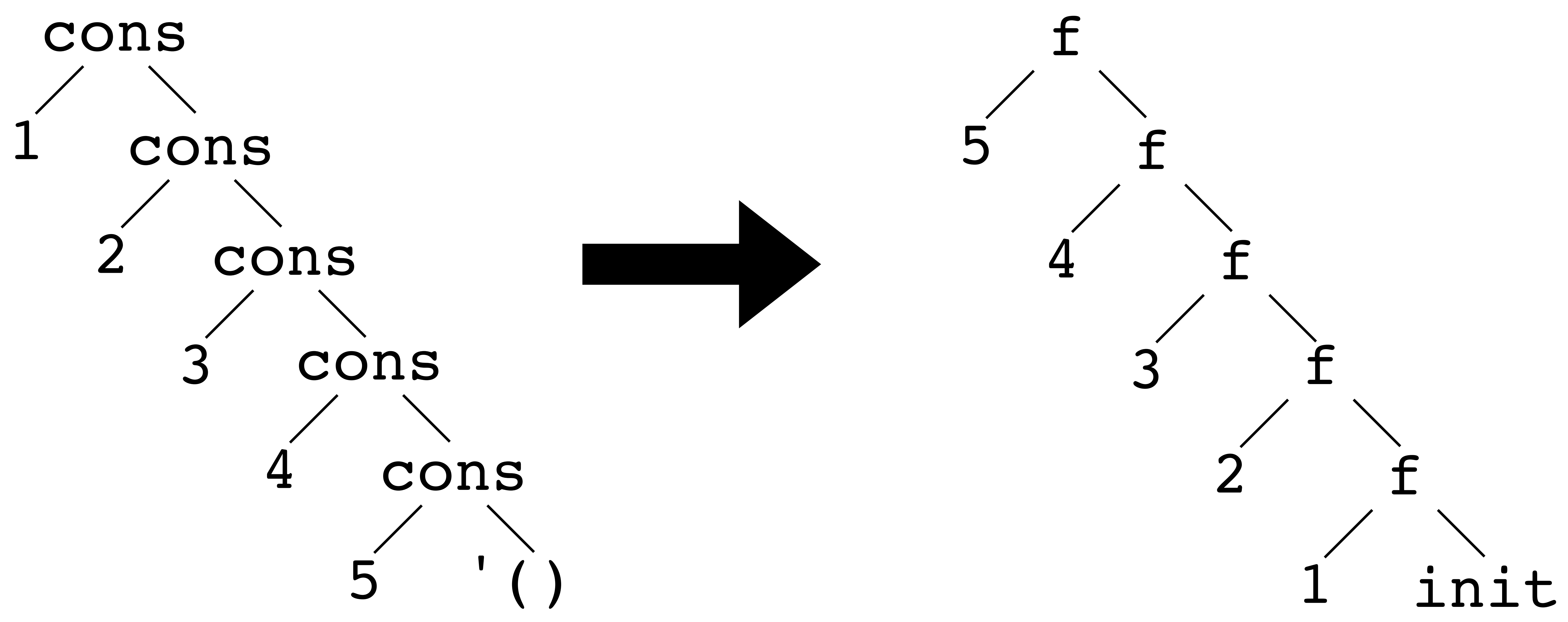
`(foldl combine base-case lst)`

`combine` has the form `(λ (head acc) …)`

We'll need to reverse the result!

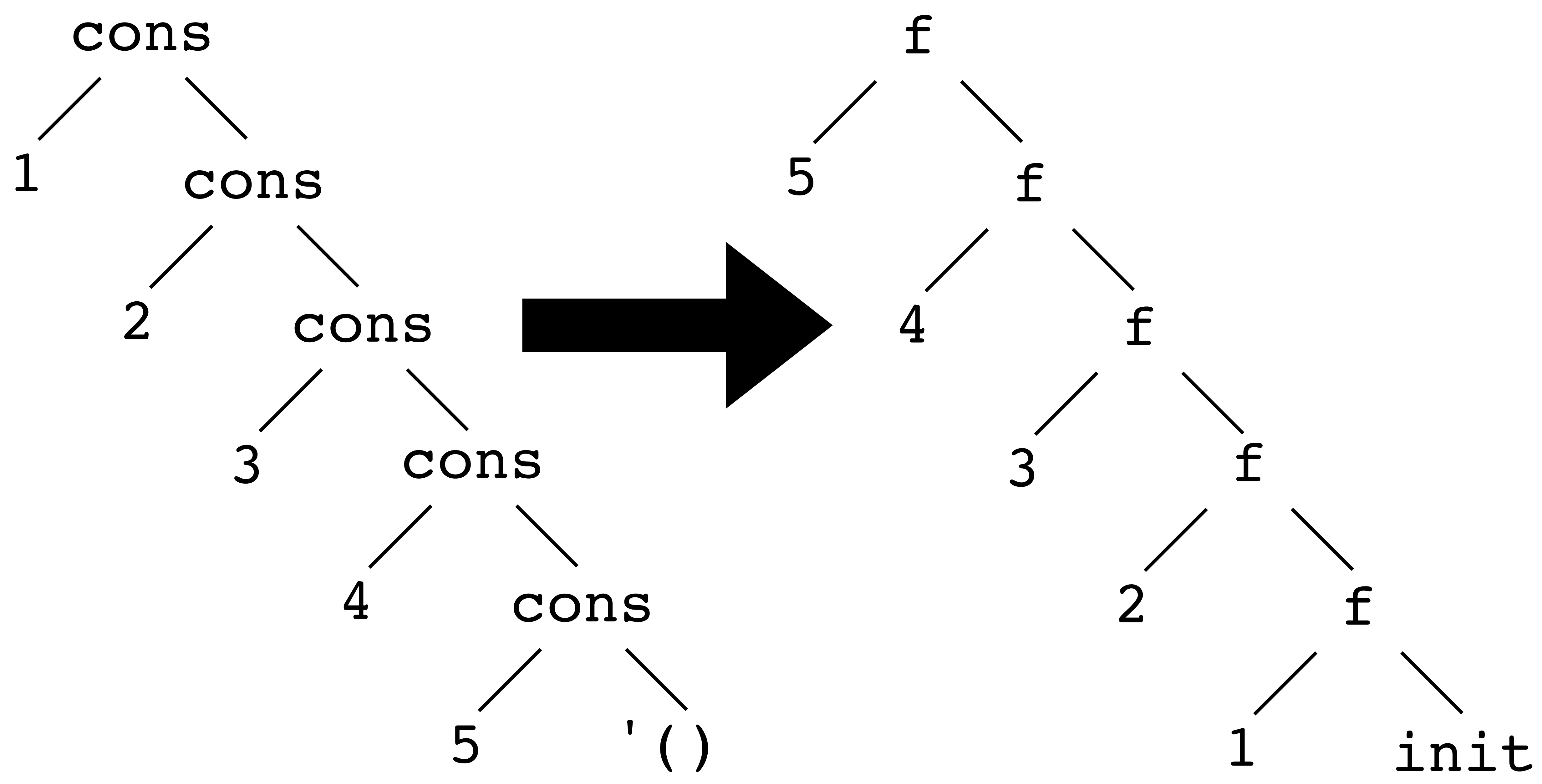
Visualizing `foldl`

`(foldl f init '(1 2 3 4 5))`

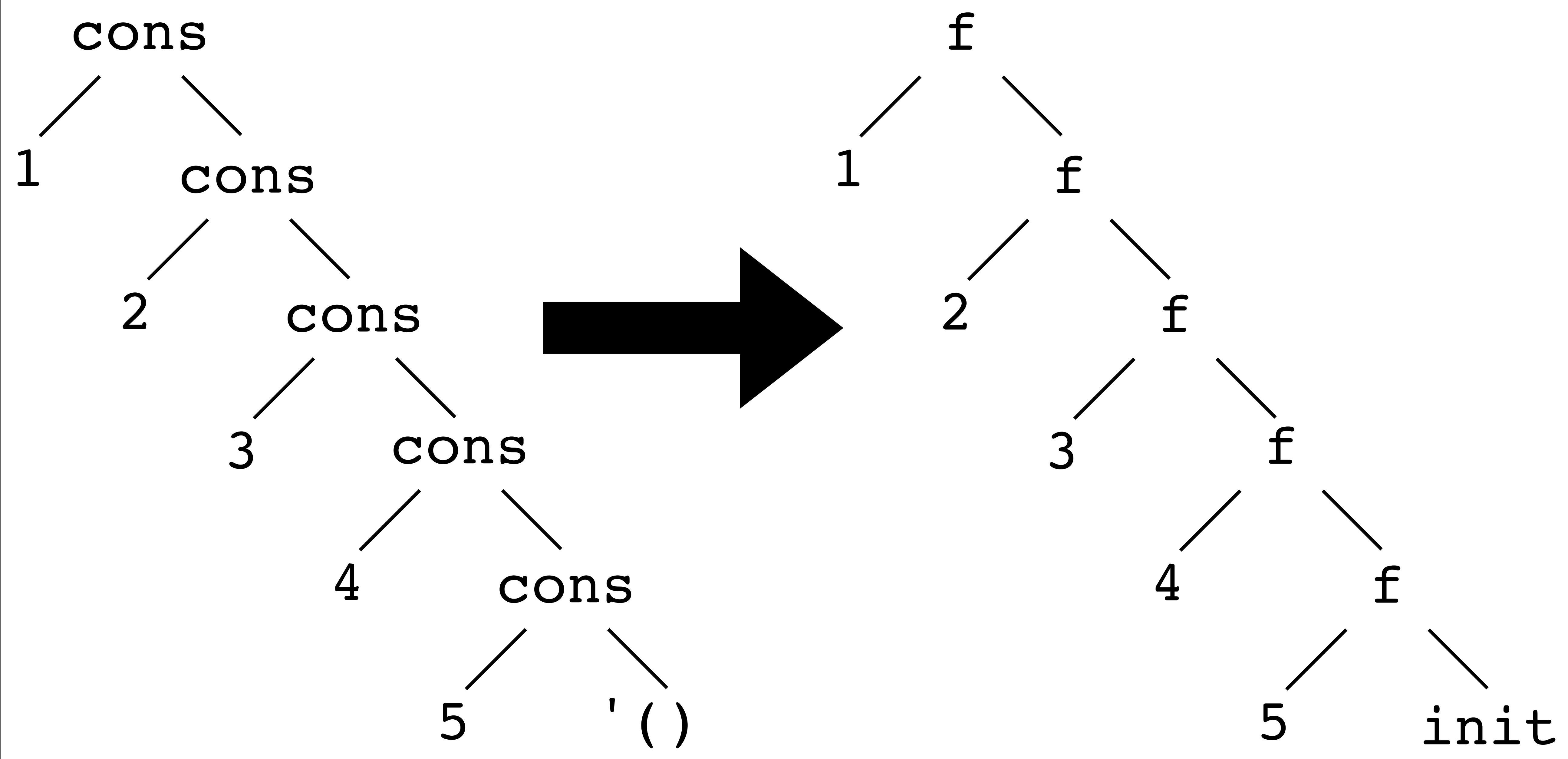


Both folds

`foldl`

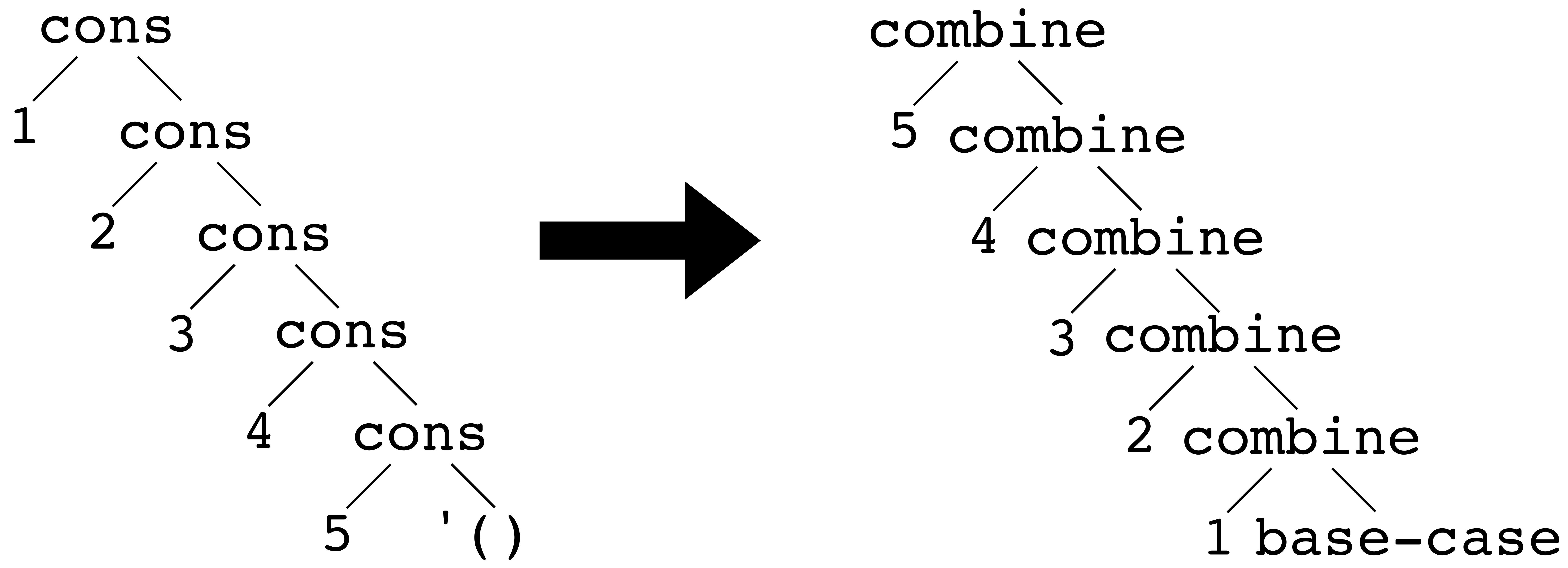


`foldr`



Let's write `foldl`

`(foldl combine base-case lst)`



Combinators and combinatory logic

An early 20th century crisis in mathematics

Russell's Paradox

Define *S* to be the set of all sets that are *not* elements of themselves

- $S = \{x \mid x \notin x\}$

Is *S* an element of *S*?

- Assume so: $S \in S \implies S \notin S$ by the definition of *S*, a contradiction
- Assume not: $S \notin S \implies S \in S$ by the definition of *S*, another contradiction!

This led to a hunt for a non-set-theoretic foundation for mathematics

- Combinatory logic (Moses Schönfinkel and rediscovered by Haskell Curry)
- Lambda calculous (Alonzo Church and others)
  - This forms the basis for functional programming!

Combinatory term

A variable (from an infinite list of possible variables)

A combinator

- One of a finite list of primitive functions; or
- A new combinator $(C\ x_1 \ldots x_n) = E$ where E is a combinatory term, all of whose variables are in the set $\{x_1, \ldots, x_n\}$

$(E_1\ E_2)$ An application of E_1 to E_2

- Application is left-associative so $(E_1\ E_2\ E_3\ E_4)$ is $(((E_1\ E_2)\ E_3)\ E_4)$

Expressing combinators in Scheme

We can represent combinators in Scheme as procedures with no free variables (i.e., every variable used in the body of the procedure is a parameter)

There are no λs in combinatory logic so no way to make new functions

However, combinatory logic does have a way to get the same effect as λ expressions

- We won't cover this, but we can convert every expression in λ calculus into combinatory logic
- λ calculus is Turing-complete (it can perform any computation) so combinatory logic is as well!

SKI combinatory logic

Three primitive combinator (and one is unnecessary!)

- The identity combinator (I x) = x
- The constant combinator (K x y) = x
  - I.e., ((K x) y) = x which you can think of as (K x) is a function that given any argument y returns x
- The substitution combinator (S f g x) = (f x (g x))
  - You can think of S as taking two functions f and g and some term x. f is applied to x which returns a function and that function is applied to the result of (g x)

Example: I is unnecessary

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))

Consider the combinatory expression (S K K x) and apply the combinator definitions from left to right

```
(S K K x) = (K x (K x))        [Substitution]
          = x                  [Constant]
```

That last one comes because (K x y) = x for any y, in particular for y = (K x)

Note that (I x) = x as well

- We say (S K K) and I are *functionally equivalent*
- Using just S and K, we can express any computation

Example: Composition combinator

(B f g x) = (f (g x))

(S (K S) K f g x)	= ((K S) f (K f) g x)	[Substitution]
	= (K S f (K f) g x)	[Associativity]
	= (S (K f) g x)	[Constant]
	= ((K f) x (g x))	[Substitution]
	= (K f x (g x))	[Associativity]
	= (f (g x))	[Constant]
	= (B f g x)	[Definition of B]

- (I x) = x
- (K x y) = x
- (S f g x) = (f x (g x))

Example: Diagonalizing combinator

(W f x) = (f x x)

Try this out on your own: (S S (S K)) = W

- Just proceed as in the previous examples, apply the rules for S and K to the combinatory term (S S (S K) f x) until you arrive at (f x x)

Expressing S, K, and I in Racket

```
(define (I x) x)

(define (K x)
  (λ (y) x))

(define (S f)
  (λ (g)
    (λ (x)
      ((f x) (g x)))))
```

Using the combinators

```
(define (identity x)
  (((S K) K) x))

(define (curry-* x)
  (λ (y)
    (* x y)))

(define (square x)
  (((S curry-*) I) x))
```

We could also define square as `((W curry-*) x)`

The Y-combinator

How do we write a recursive function?

Easy, use `define`

```
(define len
  (λ (lst)
    (cond [(empty? lst) 0]
          [else (add1 (len (rest lst)))])))
```

For the rest of this lecture, we're not going to use `(define (fun args) …)`

How do we write a recursive function?

(without using define)

Easy, use `letrec`

```
(letrec ([len
          (λ (lst)
            (cond [(empty? lst) 0]
                  [else (add1 (len (rest lst)))]))])
  len)
```

Recall, this binds length to our function `(λ (lst) …)` in the body of the `letrec`

This expression returns the procedure bound to `len` which computes the length of its argument

How do we write a recursive function?

(just using anonymous functions created via λs)

Less easy, but let's give it a go!

```
(λ (lst)
  (cond [(empty? lst) 0]
        [else (add1 (??? (rest lst)))]))
```

We need to put something in the recursive case in place of the `???` but what?

If we replace the `???` with
`(λ (lst) (error "List too long!"))`
we'll get a function that correctly computes the length of empty lists, but fails with nonempty lists

Put the function itself there?

```
(λ (lst)
  (cond [(empty? lst) 0]
        [else (add1 ((λ (lst)
                        (cond [(empty? lst) 0]
                              [else (add1 (??? (rest lst)))]))
                     (rest lst)))]))
```

Not a terrible attempt, we still have ???, but now we can compute lengths of the empty list and a single element list.

Maybe we can abstract out the function

```
(λ (len)
  (λ (lst)
    (cond [(empty? lst) 0]
          [else (add1 (len (rest lst)))])))
```

This isn't a function that operates on lists!

It's a function that takes a function `len` as a parameter and returns a closure that takes a list `lst` as a parameter and computes a sort of length function using the passed in `len` function

make-length

```
(define make-length
  (λ (len)
    (λ (lst)
      (cond [(empty? lst) 0]
            [else (add1 (len (rest lst)))]))))
```

This is the same function as before but bound to the identifier `make-length`

- The orange text is the body of `make-length`
- The purple text is the body of the closure returned by `(make-length len)`

```
(define L0 (make-length (λ (lst) (error "too long"))))
```

- `L0` correctly computes the length of the empty list but fails on longer lists

make-length

```
(define make-length
  (λ (len)
    (λ (lst)
      (cond [(empty? lst) 0]
            [else (add1 (len (rest lst)))]))))

(define L0 (make-length (λ (lst) (error "too long"))))
(define L1 (make-length L0))
(define L2 (make-length L1))
(define L3 (make-length L2))
```

- `Ln` correctly computes the length of lists of size at most n
- We need an `L∞` in order to work for all lists
- `(make-length length)` would work correctly, but that's cheating!

Enter the Y combinator

Y is a "fixed-point combinator"

If `f` is a function of one argument, then `(Y f) = (f (Y f))`

```
(Y make-length)
=> (make-length (Y make-length))
=> (λ (lst)
     (cond [(empty? lst) 0]
           [else (add1 ((Y make-length) (rest lst)))]))
```

This is precisely the length function: `(define length (Y make-length))`

How is this length?

How is this length?

Let's step through applying our length function to '(1 2 3)

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
=> (add1 (add1 (length '(3)))) ; lst is bound to '(3)
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
=> (add1 (add1 (length '(3)))) ; lst is bound to '(3)
=> (add1 (add1 (cond […][else (add1 …)])))
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
=> (add1 (add1 (length '(3)))) ; lst is bound to '(3)
=> (add1 (add1 (cond […][else (add1 …)])))
=> (add1 (add1 (add1 (length '())))) ; lst is bound to '()
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
=> (add1 (add1 (length '(3)))) ; lst is bound to '(3)
=> (add1 (add1 (cond […][else (add1 …)])))
=> (add1 (add1 (add1 (length '())))) ; lst is bound to '()
=> (add1 (add1 (add1 (cond [(empty? lst) 0][…]))))
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
=> (add1 (add1 (length '(3)))) ; lst is bound to '(3)
=> (add1 (add1 (cond […][else (add1 …)])))
=> (add1 (add1 (add1 (length '())))) ; lst is bound to '()
=> (add1 (add1 (add1 (cond [(empty? lst) 0][…]))))
=> (add1 (add1 (add1 0)))
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
=> (add1 (add1 (length '(3)))) ; lst is bound to '(3)
=> (add1 (add1 (cond […][else (add1 …)])))
=> (add1 (add1 (add1 (length '())))) ; lst is bound to '()
=> (add1 (add1 (add1 (cond [(empty? lst) 0][…]))))
=> (add1 (add1 (add1 0)))
=> 3
```

How is this length?

Let's step through applying our length function to '(1 2 3)

```
(length '(1 2 3)) ; so lst is bound to '(1 2 3)
=> (cond [(empty? lst) 0]
         [else (add1 ((Y make-length) (rest lst)))])
=> (add1 (length '(2 3))) ; lst is bound to '(2 3)
=> (add1 (cond [(empty? lst) 0]
               [else (add1 ((Y make-length) (rest lst)))]))
=> (add1 (add1 (length '(3)))) ; lst is bound to '(3)
=> (add1 (add1 (cond […][else (add1 …)])))
=> (add1 (add1 (add1 (length '())))) ; lst is bound to '()
=> (add1 (add1 (add1 (cond [(empty? lst) 0][…]))))
=> (add1 (add1 (add1 0)))
=> 3
```

But wait, how can that work?

Two problems:

- We defined Y in terms of Y! It's recursive and the whole point was to write recursive anonymous functions
- `(Y f) = (f (Y f))` but then
  `(f (Y f)) = (f (f (Y f)) = (f (f (f (Y f)))) = …`
  and this will never end

Defining Y

```
(define Y
  (λ (f)
    ((λ (g) (f (g g)))
     (λ (g) (f (g g))))))
```

It's tricky to see what's going on but Y is a function of f and its body is applying the anonymous function `(λ (g) (f (g g)))` to the argument `(λ (g) (f (g g)))` and returning the result.

```
(Y foo) = ((λ (g) (foo (g g)))             ; By applying Y to foo
           (λ (g) (foo (g g))))
        = (foo ((λ (g) (foo (g g)))        ; By applying orange fun
                (λ (g) (foo (g g)))))      ;   to purple argument
        = (foo (Y foo))                    ; From definition of Y
```

Never ending computation

This form of the Y-combinator doesn't work in Scheme because the computation would never end

We can fix this by using the related Z-combinator

This is the argument to our recursive function

```
(define Z
  (λ (f)
    ((λ (g) (f (λ (v) ((g g) v))))
     (λ (g) (f (λ (v) ((g g) v)))))))
```

With this definition, we can create a length function

```
(define length (Z make-length))
```

We can use Z to make recursive functions

Given a recursive function of one variable

```
(define foo
  (λ (x) … (foo …) …))
```

we can construct this only using anonymous functions by way of Z

```
(Z (λ (foo) (λ (x) … (foo …) …)))
```

```
Factorial
(Z (λ (fact)
     (λ (n)
       (if (zero? n)
           1
           (* n (fact (sub1 n)))))))
```

What about multi-argument functions?

We can use apply!

This is the list of arguments to our recursive function

```
(define Z*
  (λ (f)
    ((λ (g) (f (λ args (apply (g g) args))))
     (λ (g) (f (λ args (apply (g g) args)))))))
```