CSCI 275: Programming Abstractions

Lecture 10: The world of folds

Spring 2025

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

Questions for the good of the group?

α and β are types. And let's say `proc` takes elements of type α and produces elements of type β (i.e. the type of `proc` is $\alpha \rightarrow \beta$).

`When` calling `(map proc lst)`, what is the type of `lst`? What is the type of `map`'s return?

A. List of β, List of β
B. List of α, List of α
C. List of α, list of β
D. List of β, List of α
E. Something else

Review: map

Applies a procedure to each element of a list

α and β are types

`(map proc lst)`

`proc :` $\alpha \rightarrow \beta$

`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 (lambda (n b) (if b (- n) n))
       '(5 #t))
```

Even *more* abstractions, and thus tools in our toolbox

Lots of similarities between functions

(sum lst)

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

(length lst)

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

(map proc lst)

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

Even for functions that don't immediately look like they fall into the pattern…

(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)))]))
```

Even for functions that don't immediately look like they fall into the pattern…

`(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)))]))
```

We can rewrite them 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!

```
(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

What kind of function is `combine`?
(input type to output type)

A. combine: $\alpha \times \beta \rightarrow \alpha$

B. combine: $\alpha \times \beta \rightarrow \beta$

C. combine: $\beta \times \alpha \rightarrow \alpha$

D. combine: $\beta \times \alpha \rightarrow \beta$

```
(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)`

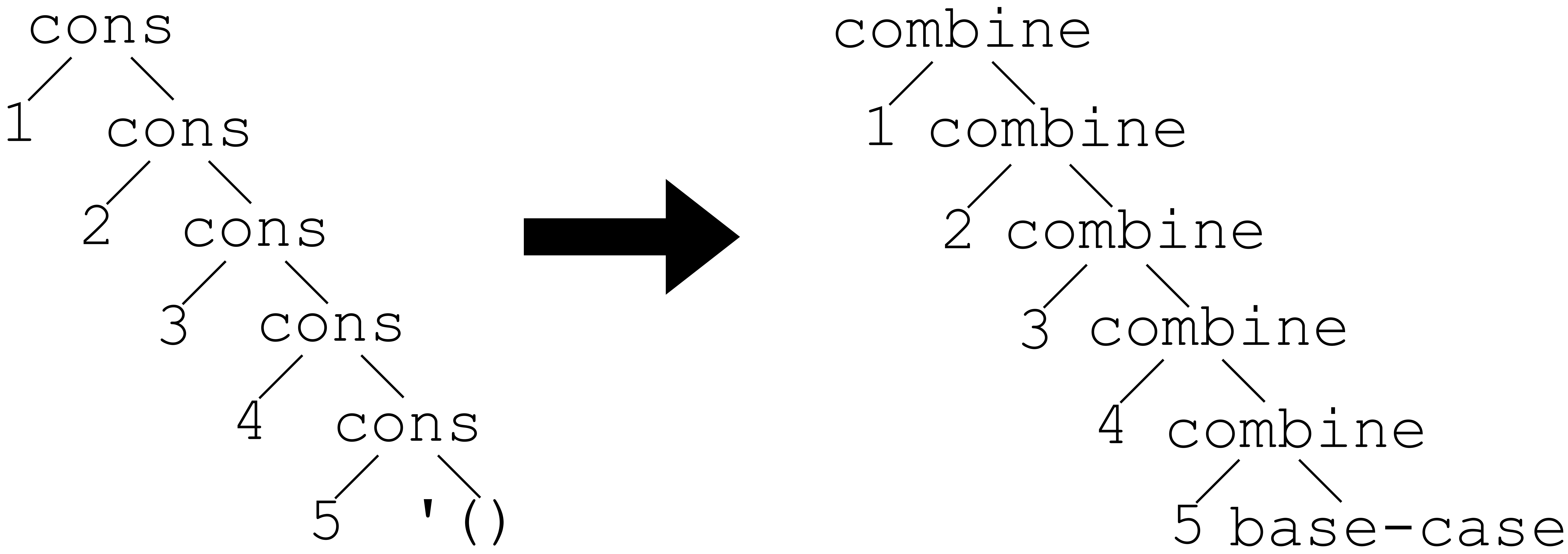
`combine`: $\alpha \times \beta \to \beta$
`base-case`: β
`lst`: `list of` α
`foldr`: $(\alpha \times \beta \to \beta) \times \beta \times (\texttt{list of } \alpha) \to \beta$

Elements of `lst` = $(x_1\ x_2\ \ldots\ x_n)$ 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})
⋮
z_1 = (combine x_1 z_2)

Abstraction: fold right

`(foldr combine base-case lst)`



Possible input `lst`

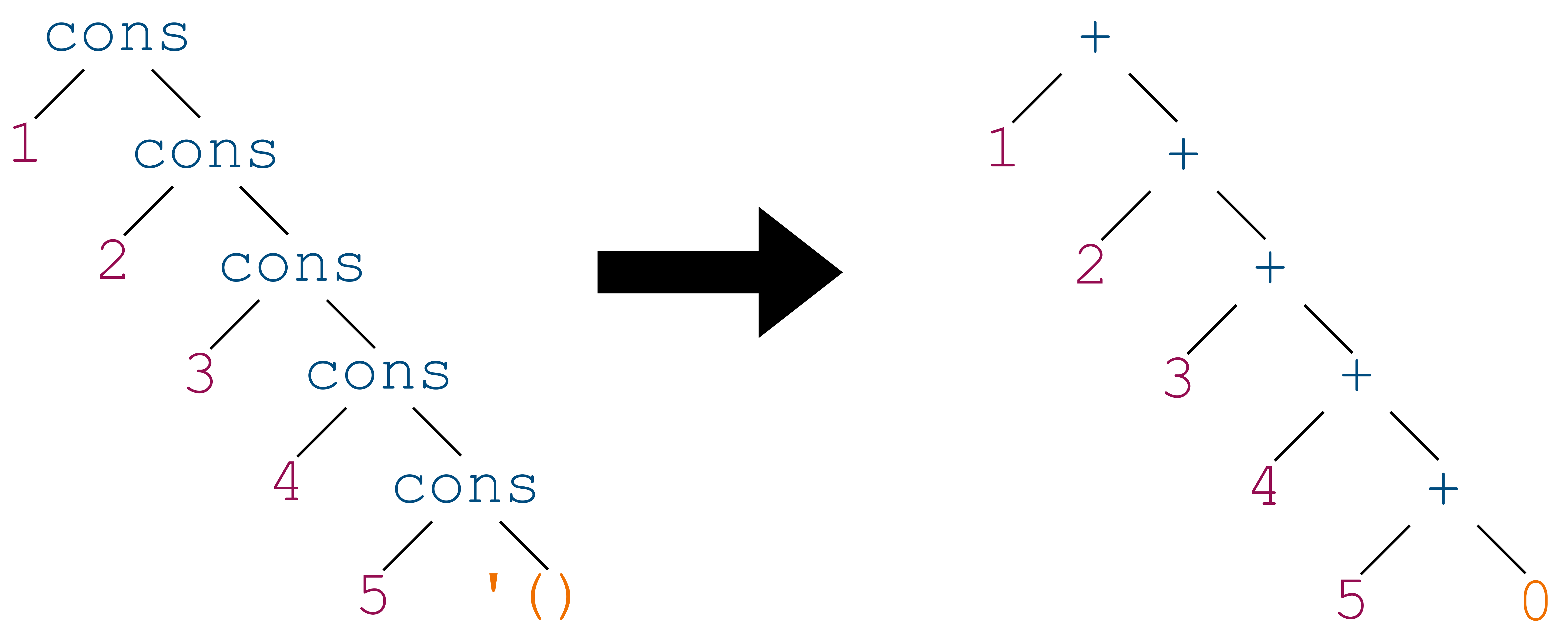
Executing `foldr`

sum as a fold right

`(foldr combine base-case lst)`

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

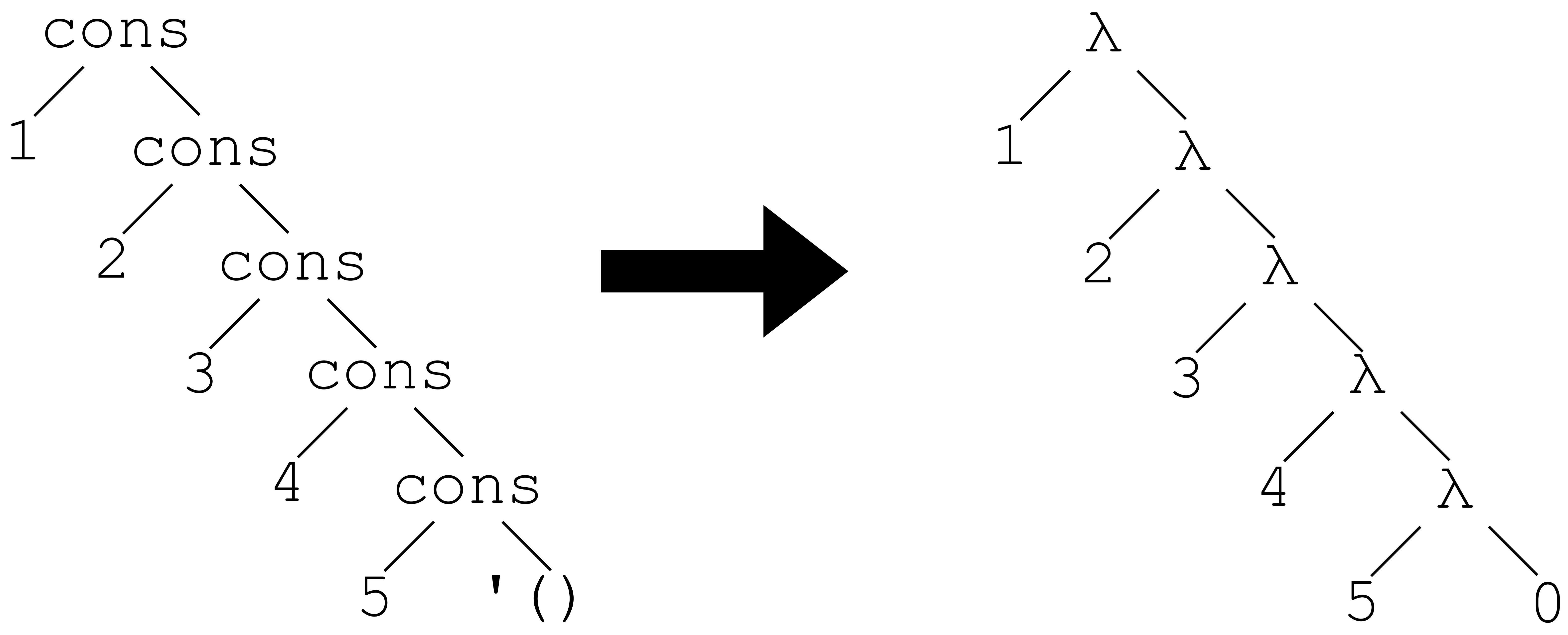
combine: number × number → number
base-case: number
lst: list of number



length as a fold right

`(foldr combine base-case lst)`

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



`map` as fold right

`(foldr combine base-case lst)`

```
(define (map proc lst)
  (foldr (lambda (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 (lambda (head result)
           (if (equal? x head)
               result
               (cons head result)))
         empty
         lst))
```

x: α

combine: α × (list of α) → list of α

base-case: list of α

lst: list of α

remove*: α × (list of α) → list of α

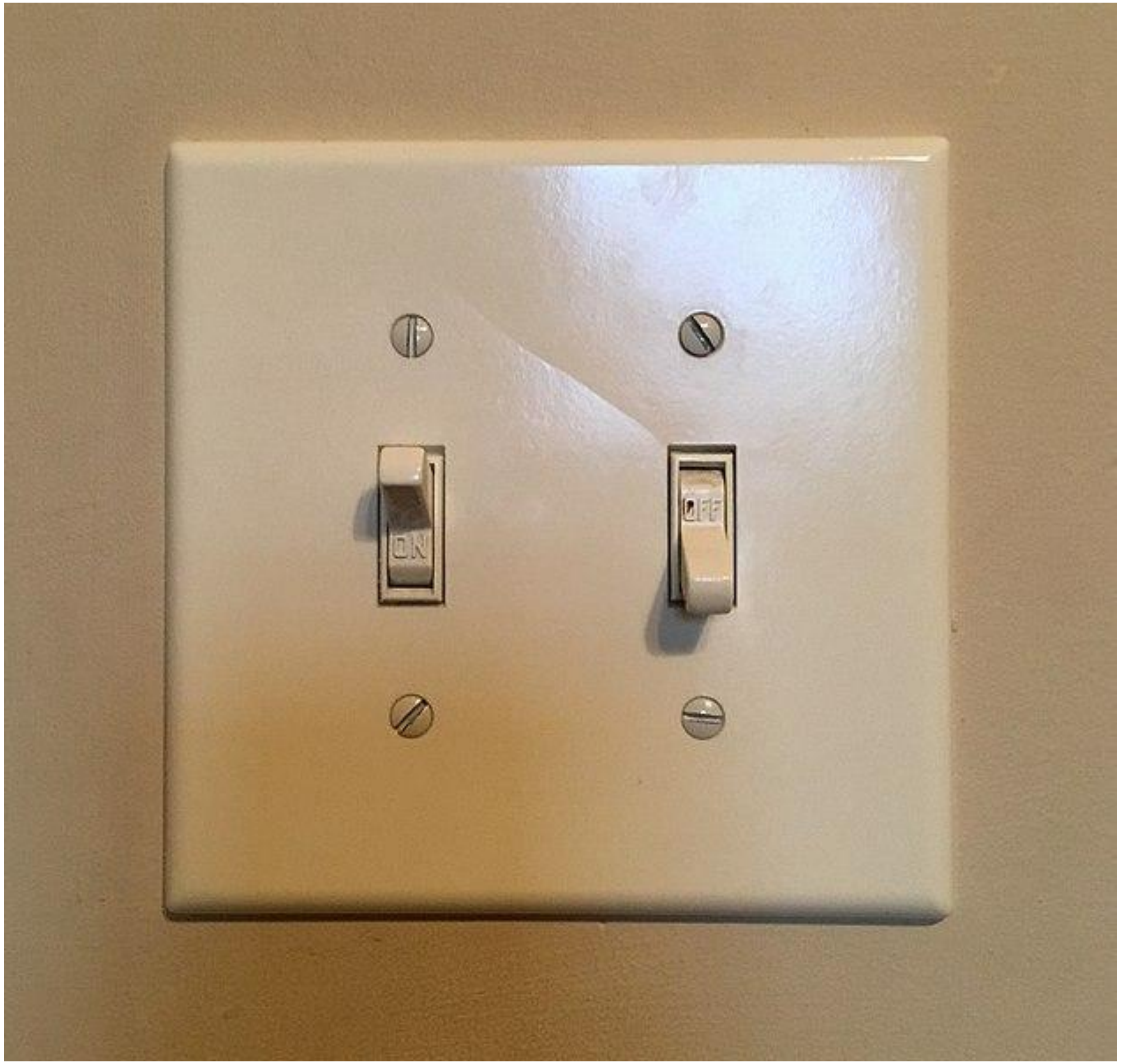
Consider the procedure

```
(foldr (lambda (str num)
              (+ num (string-length str)))
           0
           '("red" "green" "blue"))
```

What does this do?

A. Multiplies all the string lengths
B. Counts number of elements in the list
C. Sums all the string lengths
D. Error

Example: a light switch "state machine"



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

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 (no higher order needed!)

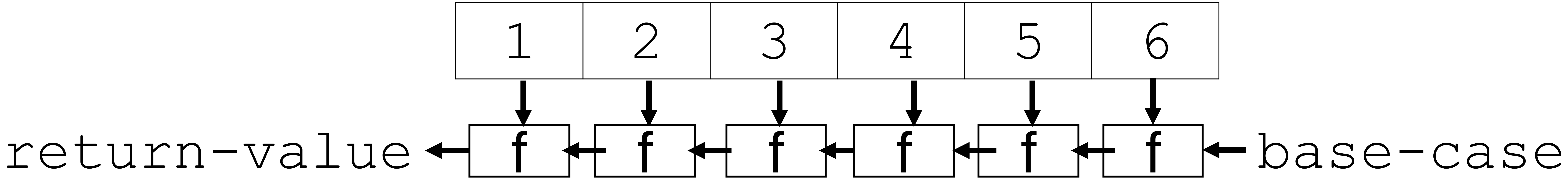
Write a `(state-after actions)` that returns the state of the light assuming it's initially `'off` and the actions in the list `actions` are performed in order

- Use `foldr!`
- Be careful about the order:

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

Takeaway from state machine example

`foldr` really is fold *right*



Next Up

Readings do continue!

Homework 2 is live, due Friday at 11:59pm via GitHub

- Feel free to use whatever structures you’d like to solve it (higher order not required, HW3/4 they will be!)