foldr

CS 5010 Program Design Paradigms
"Bootcamp"

Lesson 6.4

Introduction

- In this lesson, we will explore another common pattern in functions defined by the list template.
- We will generalize this to a function called **`foldr`**.
- We will visualize how **`foldr`** works, and show an important application area.

Learning Objectives

- At the end of this lesson you should be able to:
  - describe, recognize, and use the **`foldr`** pattern.

What else could be different?

```
;NumberList -> NumberList
(define (add-1-to-each lon)
  (cond
    [(empty? lon) empty]
    [(else (cons
             (add1
               (first lon))
             (add1-to-each
               (rest lon))))]))
```

```
;EmployeeList -> ListOfString
(define (extract-names lop)
  (cond
    [(empty? lop) empty]
    [else (cons
            (Employee-name
              (first lop))
            (extract-names
              (rest lop)))]))
```

Here is the example we used to introduce map. In this example, both of the brown functions are **cons**, but in some other function there could be something else in that position.

Another example

```
;; NumberList -> Number
 (define (sum lon)
   (cond
     [(empty? lon) 0]
     [else (+
             (first lon)
             (sum
              (rest lon)))]))
```

```
;; NumberList -> Number
 (define (product lon)
   (cond
     [(empty? lon) 1]
     [else (*
             (first lon)
             (product
              (rest lon)))]))
```

Both these functions take a list of numbers and return a number. **sum** returns the sum of the elements of the given list. **product** returns the product of the elements of the given list.

These functions are just alike, except for the differences marked in red and green.

Let's generalize these

- **sum** and **product** can be generalized to a function we call **foldr**, with two new arguments: one called **fcn**, for the function in the green position, and one called **val**, for the value in the red position. The strategy for **foldr** is using the template for XList on its list argument.
- Our original **sum** and **product** functions can be recreated by supplying **+** and **0**, or ***** and **1**, as the two arguments. The strategy for these new versions of **sum** and **product** is "Use HOF foldr on ...".
- The name **foldr** is a standard name for this function, so that is the name we will use. **foldr** is already defined in ISL, so you don't need to write out the definition.
- Let's look at the code:

Create two new arguments for the two differences.

We call this "foldr" (we'll explain the name later)

```
(define (foldr fcn val lon)
   (cond
     [(empty? lon) val]
     [else (fcn
               (first lon)
               (foldr fcn val (rest lon)))]))

;; strategy: Use HOF foldr on lon
(define (sum lon) (foldr + 0 lon))
(define (product lon) (foldr * 1 lon))
```

This is predefined in ISL, so you don't need to write out this definition

What is the purpose statement?

```
;; foldr : (X Y -> Y) Y XList -> Y
;; RETURNS: the result of applying f on the
;; elements of the given list
;; from right to left, starting with base.
;; (foldr f base (list x_1 ... x_n))
;;   = (f x_1 ... (f x_n base))
```

What is the contract for foldr?

Based on our two examples we might guess the following contract for foldr: Here is one guess for the contract for **foldr**, based on our two examples:

```
foldr :
  (Number Number -> Number) Number NumberList
    -> Number
```

This works, because **+** and ***** both have contract **`(Number Number -> Number)`**, and 0 and 1 are both numbers.

What is the contract for **foldr**?

But there is nothing in the definition of **foldr** that mentions numbers, so **foldr** could work at contract

```
(X X -> X) X XList  ->  X
```

that is, you could use **foldr** at

```
(Boolean Boolean -> Boolean)
Boolean
BooleanList
-> Boolean
```

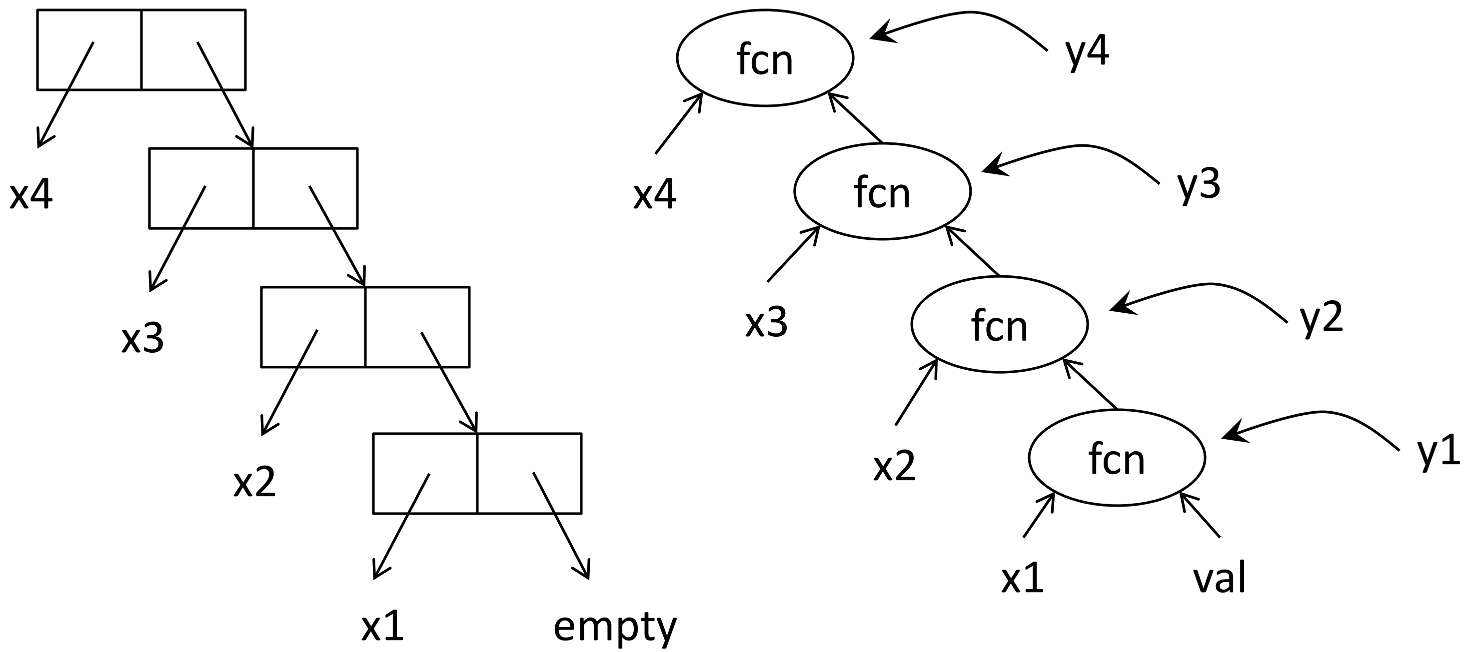
The 3 arguments to **foldr**

or

```
(Employee Employee -> Employee)
Employee
EmployeeList
-> Employee
```

The 3 arguments to **foldr**

Let's watch **`foldr`** compute on this list



Step through the animation to watch the computation of
`(foldr fcn val (list x4 x3 x2 x1))`

What can we learn from this?

- The base value **`val`** is a possible 2nd argument to **`fcn`**.
- The result of **`fcn`** becomes a 2nd argument to **`fcn`**.
- So this will work as long as
  - **`val`**,
  - the 2nd argument to **`fcn`**,
  - and the result of **`fcn`**

  are all of the same type.
- So **`fcn`** must satisfy the contract **`(X Y -> Y)`** for some **`X`** and **`Y`**.

What else can we learn?

- The elements of the list become the first argument to **`fcn`**.
- So if **`fcn`** satisfies the contract **`(X Y -> Y)`**, then the list must be of type **`XList`**.
- So the contract for foldr is:

```
foldr : (X Y -> Y) Y XList -> Y
```

The contract for foldr (again!)

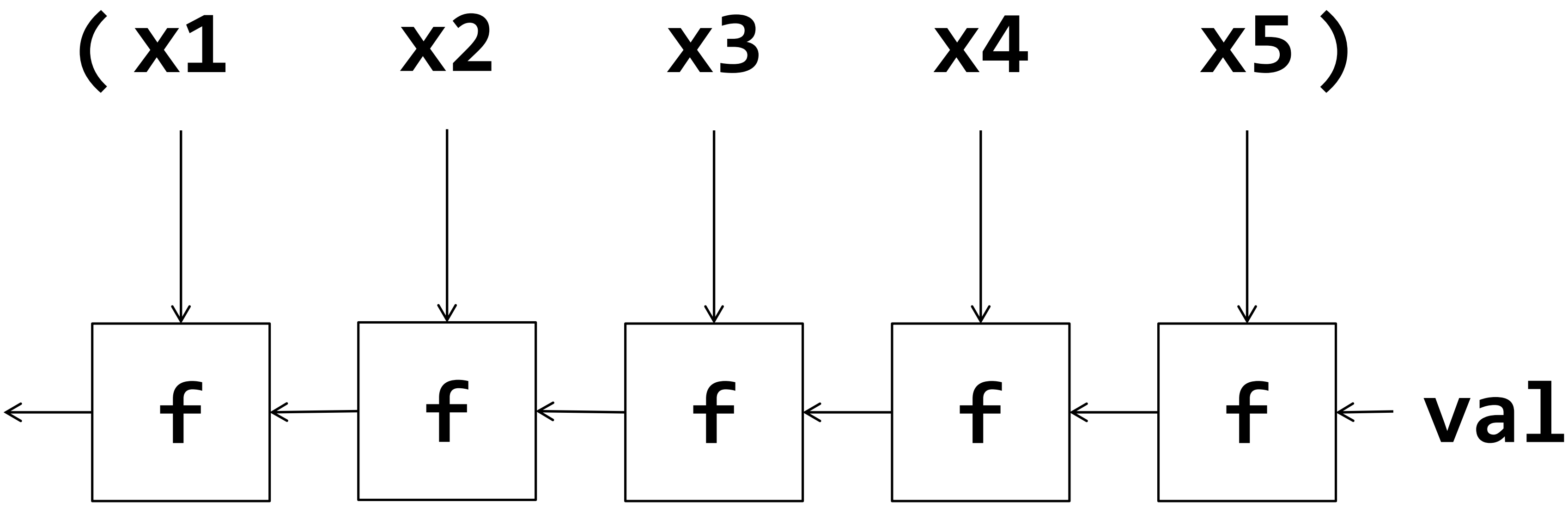
- The contract for foldr is

```
foldr : (X Y -> Y) Y XList -> Y
```

- So **`foldr`** takes 3 arguments:
  - a combiner function that satisfies the contract
    `(X Y -> Y)`
  - a base value of type **Y**
  - and a list of X's.
- And it returns a value of type **Y**.

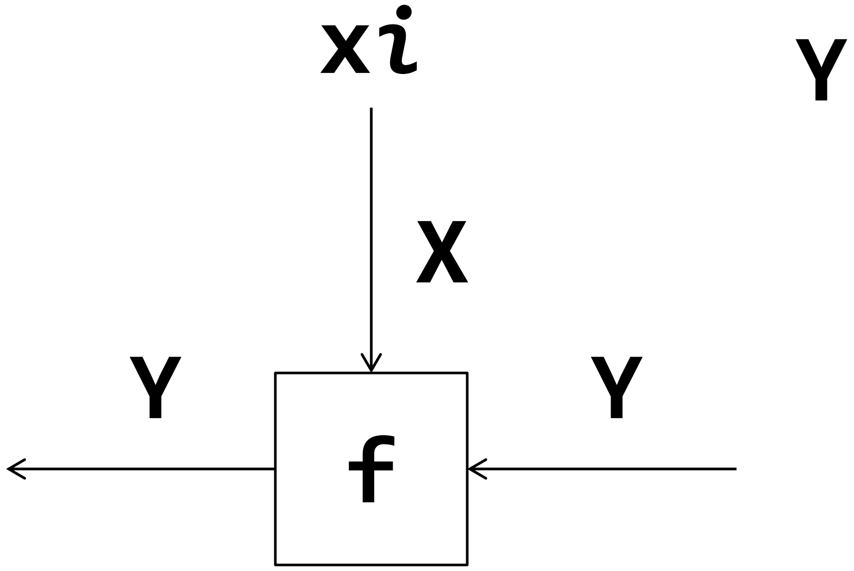
Another picture of **foldr**

Here's another visualization of foldr that you may find helpful.

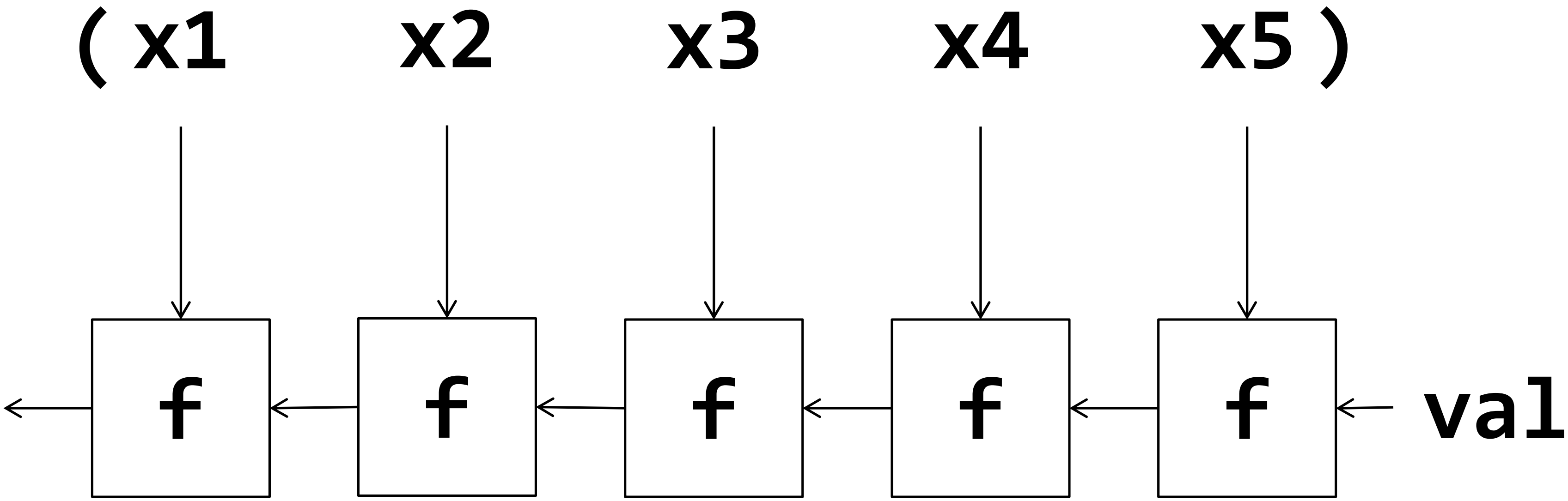


`(foldr f val (list x1 ... x5))`

What kind of data is on each arrow?



We can think of **foldr** as starting with the base value **val**, and putting it through a pipeline of **f**'s, where each **f** also takes one of the **x**'s as an input. The **x**'s are taken right-to-left, which is why it is called **foldr** .



(foldr f a (list x1 ... x5))

Another example:

```
;; strategy: combine simpler functions
(define (add1-if-true b n)
  (if b (+ n 1) n))

;; strategy: Use HOF foldr on lob
(define (number-of-trues lob)
  (foldr add1-if-true 0 lob))
```

Or even better:

```
;; strategy: Use HOF foldr on lob
(define (number-of-trues lob)
  (local ((define (add1-if-true b n)
             (if b (+ n 1) n)))
    (foldr add1-if-true 0 lob)))
```

What is the contract for **add1-if-true** ? At what contract is **foldr** being used in this example? What is returned by **count-trues** ? Try to answer these questions before proceeding to the next slide.

What are the contracts?

add1-if-true : Boolean Number -> Number

In general:

foldr : (X Y -> Y) Y XList -> Y

In this case, **X** = Boolean and **Y** = Number, so we are using **foldr** at the contract

```
  (Boolean Number -> Number)
      Number BooleanList -> Number
```

and therefore

count-trues : BooleanList -> Number

Local functions need contracts and purpose statements too

```
(define (count-trues lob)
  (local (; add1-if-true : Boolean Number -> Number
          ; RETURNS: the number plus 1 if the boolean is
          ; true, otherwise returns the number unchanged.
          (define (add1-if-true b n) (if b (+ n 1) n)))
    (foldr add1-if-true 0 lob)))
```

- They count as help functions, so they don't need separate tests.

Local functions need their deliverables, too. They count as help functions, so they don't need separate tests. If they are complicated enough to need examples or tests, then you should make them independent functions with a full set of deliverables.

The whole thing (less examples and tests)

```
;; count-trues : BooleanList -> Number
;; RETURNS: the number of trues in the given list of booleans.
;; STRATEGY: Use HOF foldr on lob
(define (count-trues lob)
  (local (; add1-if-true : Boolean Number -> Number
          ; RETURNS: the number plus 1 if the boolean is
          ; true, otherwise returns the number unchanged.
          (define (add1-if-true b n)
                   (if b (+ n 1) n)))
    (foldr add1-if-true 0 lob)))
```

HOFs can help you write code for trees, too

```
;; grandchildren : Person -> PersonList
;; STRATEGY: Use template for Person on p
(define (grandchildren p)
  (all-children (person-children p)))

;; all-children : PersonList -> PersonList
;; STRATEGY: Use HOF map on ps
(define (all-children ps)
  (foldr append empty
    (map person-children ps)))
```

Remember **grandchildren**, from Lesson 5.2. Here's a version that uses **foldr** in **all-children**.

A **PersonList** is a list, so we can use our list abstractions in the definition of **all-children**.

Compare this version of all-children with the one in Lesson 5.2. Do you see the connection?

Here is **all-descendants**, rewritten with **foldr**.

```
;; Person -> PersonList
;; STRATEGY: Use template for Person on p
(define (person-descendants p)
  (append
   (person-children p)
   (all-descendants (person-children p))))

;; PersonList -> PersonList
;; STRATEGY: Use HOF map followed by foldr
(define (all-descendants ps)
  (foldr append empty
    (map person-descendants ps)))
```

The functions are still mutually recursive.

mapreduce

```
;; mapreduce : (Y Y -> Y) Y (X -> Y) XList -> Y
;; GIVEN: f v g (list x1 ... xn))
;; RETURNS:
;;  (f (g x1) (f (g x2) (f (g x3) ... v)))
(define (mapreduce f v g lst)
  (foldr f v (map g lst)))
```

Many problems can be stated in this form, with f associative, and v the identity of f.

Mapreduce (cont'd)

- When f is associative and v is the identity of f, we can write f infix, i.e.:
  - We write $x\ [f]\ y$ instead of **`(f x y)`**
- Then we can write the result of mapreduce as

$$(g\ x1)\ [f]\ (g\ x2)\ [f]\ \ldots\ [f]\ (g\ xn)$$

Tiny Example

```
;; BoolList -> NonNegInt
;; RETURNS: the number of trues in the given list of
;; Booleans.

(define (number-of-trues lob)
  (let ((mapper (lambda (b) (if b 1 0))))
    (mapreduce + 0 mapper lob)))
```

+ is associative, and 0 is its identity

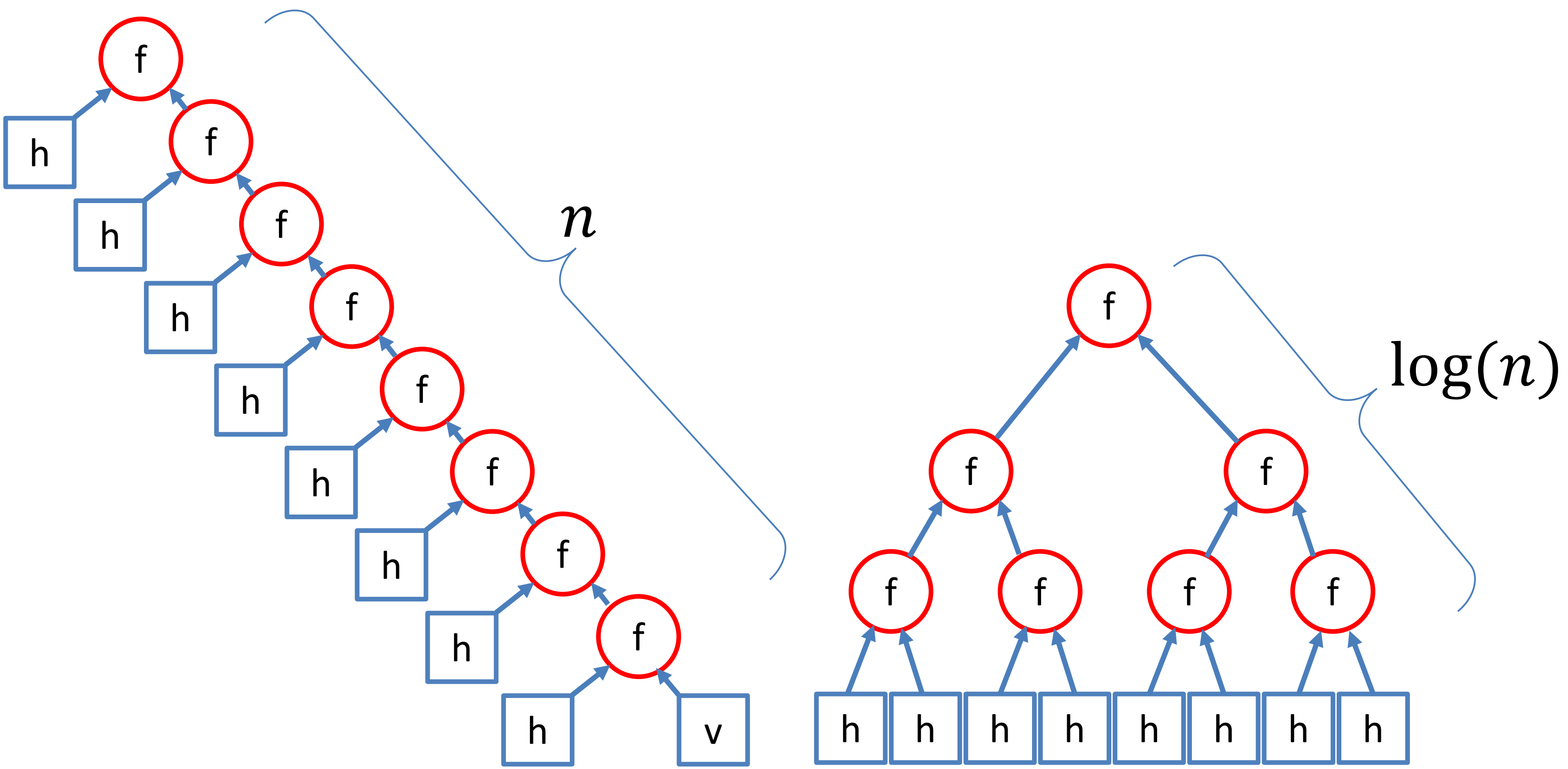
Many database tasks can be expressed in this form

- SQL operations:
  - Count
  - Filter
  - ALL
  - EXISTS

Why this wins

- When **f** is associative, and **v** is its identity, can turn the calls to **f** into a tree and do them in parallel on a server farm!
- For a data set of size *n*, this reduces the processing time from *n* to log(*n*).
- Here is a picture:

From linear time to logarithmic



You could do some of this in parallel

- Divide the subtrees across several different processors or clusters…

Where does the data come from?

- The data might not be a list.
- It might be data extracted from a large database.
- So your application would have 2 parts
  - some SQL to extract a table full of data (like "map")
  - the function you want to reduce the data with.
- The SQL insulates your application from the physical layout of the DB; the SQL query optimizer can probably get your data out of the DB fast.
- mapreduce systems (like Hadoop or Spark) allow you to configure the 'reduce' phase to make use of the available hardware.

Summary

- You should now be able to:
  - describe, recognize, and use the **`foldr`** pattern.
  - state the contracts for **`ormap`**, **`andmap`**, and **`filter`** and **`foldr`**, and use them appropriately.
  - combine these functions using higher-order function combination.

Next Steps

- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 6.4
- Go on to the next lesson