Foldr and Foldl

CS 5010 Program Design Paradigms

Lesson 7.5

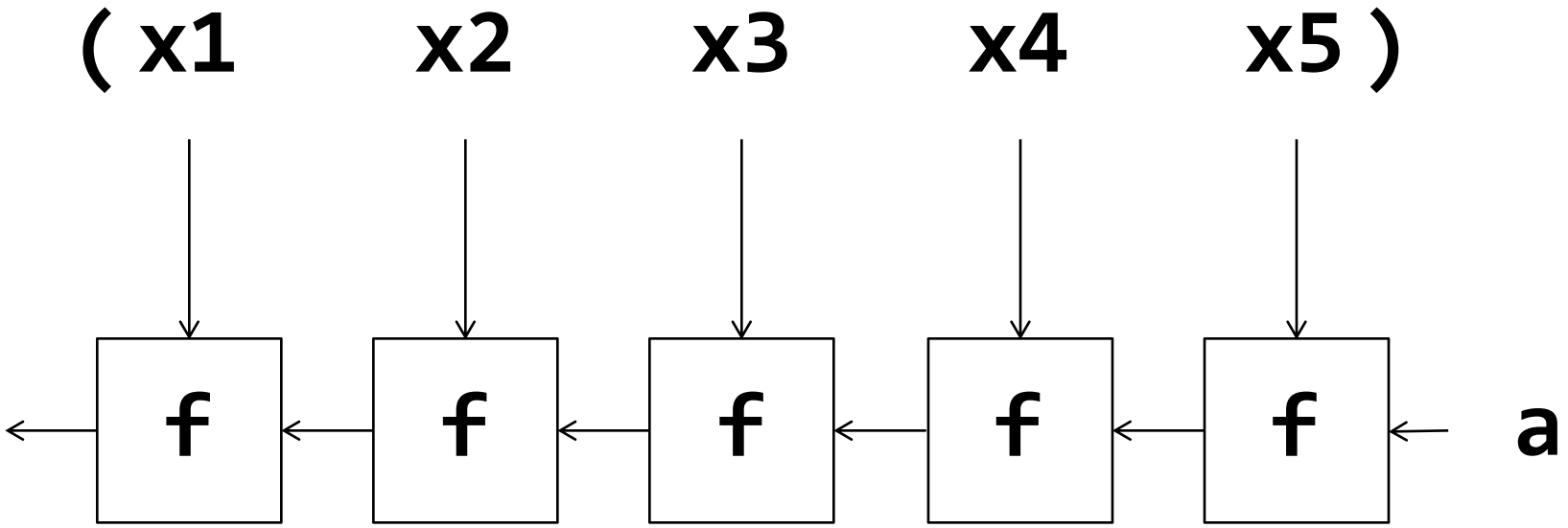
Lesson Outline

- Look more closely at foldr
- Introduce foldl: like foldr but "in the other direction"
- Implement foldl using a context variable
- Look at an application

Learning Objectives

- At the end of this lesson you should be able to:
  - explain what **foldr** and **foldl** compute
  - explain the difference between **foldr** and **foldl**
  - explain why they are called "fold right" and "fold left"
  - use **foldl** in a function definition

Foldr: the general picture




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

Another picture of foldr

The textbook says:

```
;; foldr : (X Y -> Y) Y ListOfX -> Y
;; (foldr f base (list x_1 ... x_n))
;;   = (f x_1 ... (f x_n base))
```

This may be clearer if we write the combiner in infix:
eg **(x - y)** instead of **(f x y)** :

```
(foldr – a (list x1 ... xn)) =
  x1 – (x2 – (... – (xn – a)))
```

We use – instead of +, because – is not associative. So it makes a difference which way you associate
x1 – x2 – x3 – x4

What if we wanted to associate the other way?

Instead of

x1 – (x2 – (... – (xn – a)))

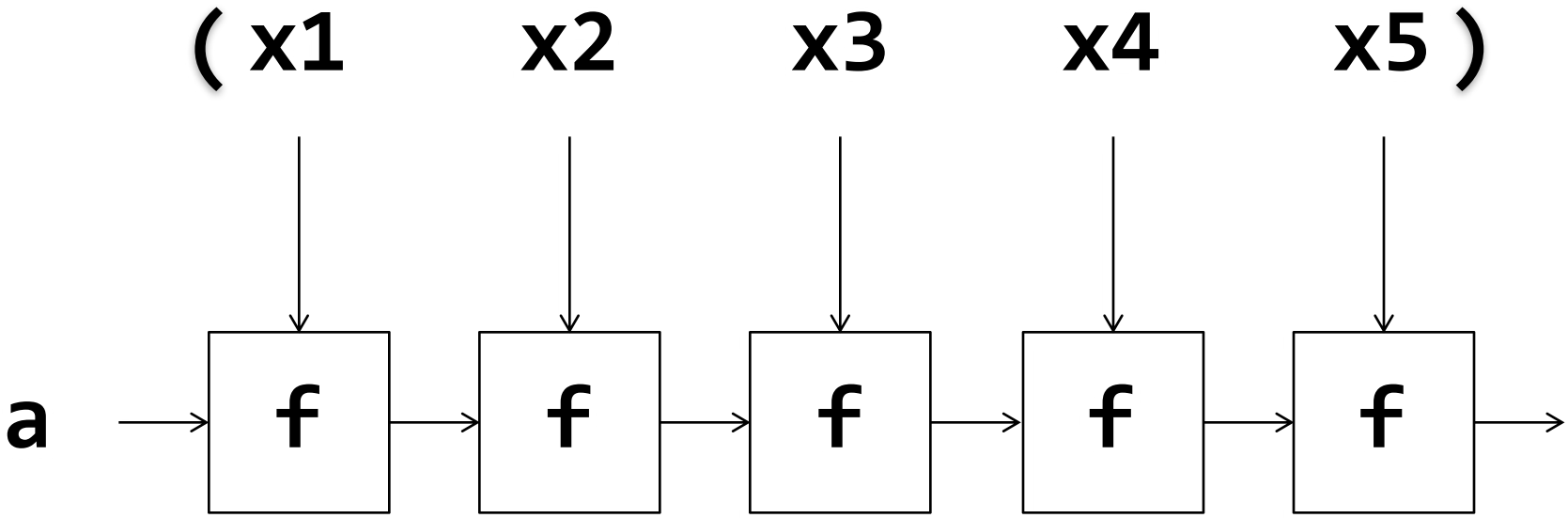
suppose we wanted

(((a – x1) – x2) ... – xn)

foldr associates its operator to the right

foldl will associate its operator to the left

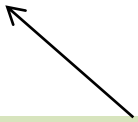
For this computation, the pipeline goes the other way




```
(foldl f a (list x1 ... x5))
```

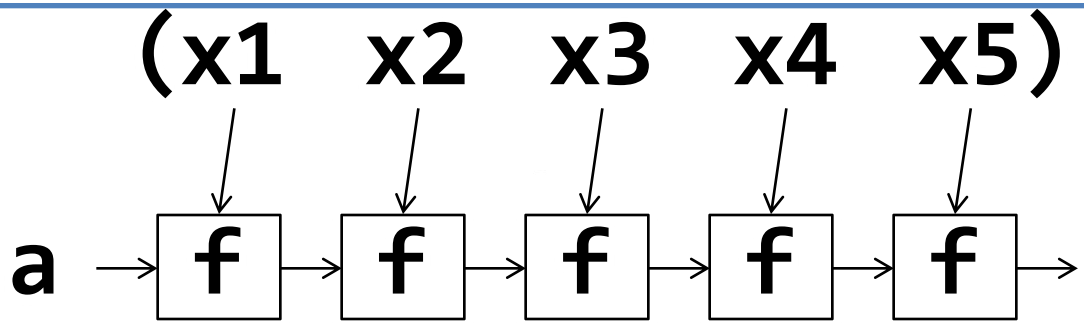
Let's write the code

```
;; We'll use the template:
(define (foldl f a lst)
  (cond
   [(empty? lst) ...]
   [else (...
          (first lst)
          (foldl ... (rest lst)))]))
```



We'll need to figure out what goes here.

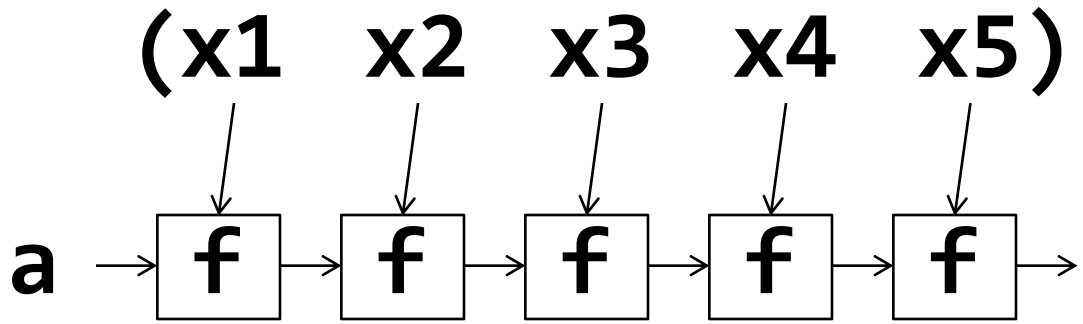
What if lst is empty?



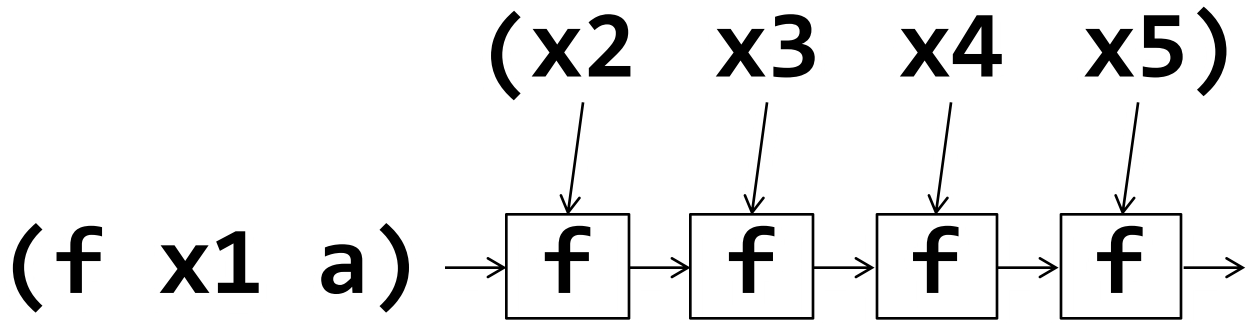
- When the list is empty, there are no stages in the pipeline, so

```
(foldl f a empty) = a
```

What if the list is non-empty?



=



So for a non-empty list

```
(foldl f a (cons x1 lst))
  = (foldl f (f x1 a) lst)
```

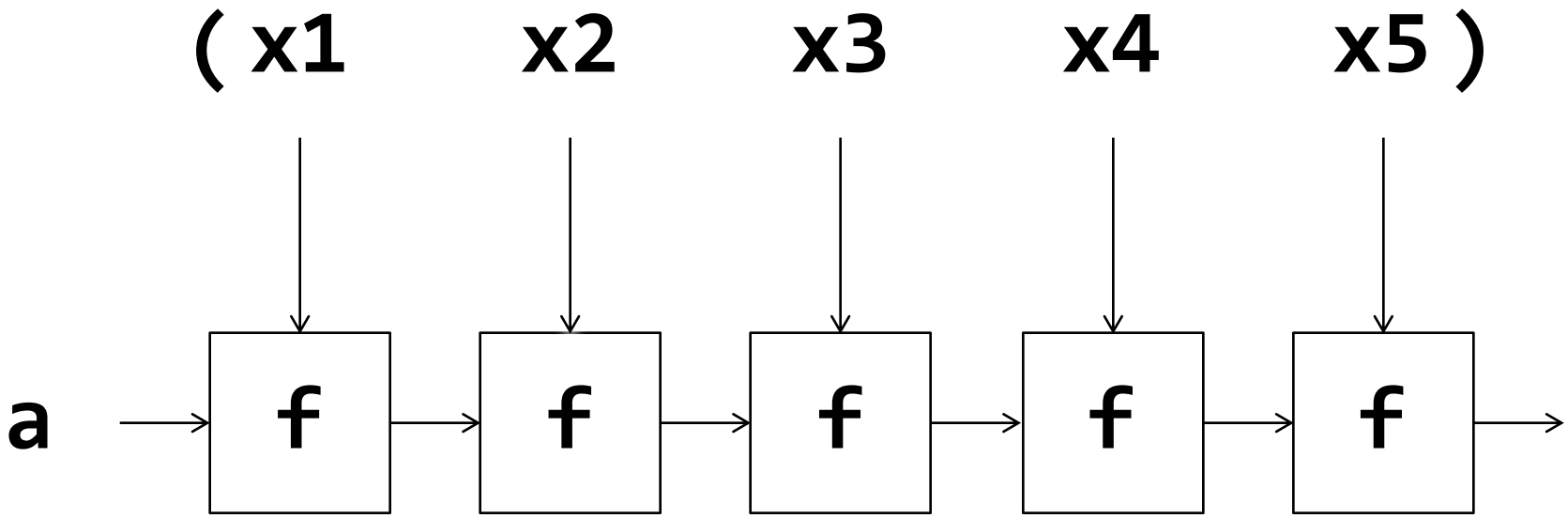
Putting this together

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

Let's do a computation

```
   (foldl - 1 (list 20 10 2))
= (foldl - 19 (list 10 2))  ;20-1 = 19
= (foldl - -9 (list 2))     ;10-19 = -9
= (foldl - 11 empty)        ;2-(-9) = 11
= 11
```

What's the contract?



This part is like foldr: We can label all the vertical arrows as X's and all the horizontal arrows as Y's, so the contract becomes

```
(X Y -> Y) Y ListOfX -> Y
```

Purpose Statement (1)

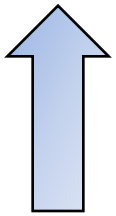
- Textbook description:

```
;; foldl : (X Y -> Y) Y ListOfX -> Y
;; (foldl f base (list x_1 ... x_n))
;;   = (f x_n ... (f x_1 base))
```

Can we describe this using an invariant?

- To do this, let's think about where we are in the middle of a computation

```
(((a – x1) – x2) x3 ... – xn)
```



- At this point, we've processed x1 and x2, and we are looking at the sublist **`(x3 ... xn)`**

Purpose Statement using invariant

```
GIVEN: a function f, a value a, and a sublist lst
WHERE: lst is a sublist of some larger list lst0
AND: a is the result of applying f to some starting
     element a0 and the elements of lst0 that are above lst
RETURNS: the result of applying f to the starting element a0
         and all the elements of lst0.
```

Here's an alternate purpose statement that describes the situation in the middle of the pipeline.

You don't have to use this purpose statement; you can use the one from the book if it is easier for you.

Let's apply this to subtraction

```
;; diff : NonEmptyListOfNumber -> Number
;; GIVEN: a nonempty list of numbers
;; RETURNS: the result of subtracting the numbers, from
;;          left to right.
;; EXAMPLE:
;; (diff (list 10 5 3)) = 2

;; We'll use the data definition
;; NELON = (cons Number ListOfNumber)
```

This was guided practice 7.1

Code, with simple purpose statement

```
(define (diff nelst)
  (diff-inner (first nelst) (rest nelst)))

;; diff-inner : Number ListOfNumber
;; RETURNS: the result of subtracting each of the numbers in lon
;; from num
(define (diff-inner num lon)
  (cond
    [(empty? lon) num]
    [else (diff-inner
            (- num (first lon))   ;; this is (f a (first lon))
                                  ;; different order of arguments
                                  ;; than foldl
            (rest lon))]))
```

Code, with fancier purpose statement

```
(define (diff nelst)
  (diff-inner (first nelst) (rest nelst)))

;; diff-inner : Number ListOfNumber
;; GIVEN: a number sofar and a sublist lon of some list lon0
;; WHERE: sofar is the result of subtracting all the numbers in
;; lon0 that are above lon.
;; RETURNS: the result of subtracting all the numbers in lon0.
(define (diff-inner sofar lon)
  (cond
    [(empty? lon) sofar]
    [else (diff-inner
            (- sofar (first lon)) ;; this is (f a (first lon))
                                  ;; different order of arguments
                                  ;; than foldl
            (rest lon))]))
```

sofar is a good name for this argument

You could use either this purpose statement or the one on the preceding slide. Both are fine.

Or using foldl

```
(define (diff nelst)
  (foldl
    (lambda (x sofar) (- sofar x))  ;; foldl wants an X Y -> Y
    (first nelst)
    (rest nelst)))
```

sofar is a good name for this argument, because it describes where the value comes from.

Another application: Simulation

```
;; simulating a process

;; Wishlist:
;; next-state : Move State -> State

;; simulate : State ListOfMove -> State
;; given a starting state and a list of
;; moves, find the final state
```

An Application: Simulation

```
;; strategy: structural decomposition on moves
(define (simulate st moves)
  (cond
    [(empty? moves) st]
    [else
      (simulate
        (next-state (first moves) st)
        (rest moves))]))
```

Or using foldl

```
(define (simulate initial-state moves)
  (foldl
   next-state
   initial-state
   moves))
```

I carefully chose the order of the arguments to make this work. If next-state took its arguments in a different order, you'd have to do the same kind of thing we did for subtraction above.

Summary

- You should now be able to:
  - explain what **foldr** and **foldl** compute
  - explain the difference between **foldr** and **foldl**
  - explain why they are called "fold right" and "fold left"
  - use **foldl** in a function definition

Next Steps

- If you have questions about this lesson, ask them on the Discussion Board
- Do Problem Set 07