MOD08: Functional Programming

Higher-order functions

Marco Gerards

25-04-2019

Practical session

- Suggested planning
  - One set each 4-hour session
  - Now: set 1 finished (1-FP.1 to 1-FP.11)
  - Tomorrow: finish set 2 (1-FP.12 to 1-FP.20)

Practical session

- Suggested planning
  - One set each 4-hour session
  - Now: set 1 finished (1-FP.1 to 1-FP.11)
  - Tomorrow: finish set 2 (1-FP.12 to 1-FP.20)
- Don’t wait with signing off; sign-off after exercises:
  - Block 1 (set 1): 1-FP.4, 1-FP.8, 1-FP.11
  - Block 1 (set 2): 1-FP.15, 1.FP.18, 1.FP.20
  - Block 2 (set 3): 2-FP.4, 1-FP.10

Practical session

- Suggested planning
  - One set each 4-hour session
  - Now: set 1 finished (1-FP.1 to 1-FP.11)
  - Tomorrow: finish set 2 (1-FP.12 to 1-FP.20)
- Don’t wait with signing off; sign-off after exercises:
  - Block 1 (set 1): 1-FP.4, 1-FP.8, 1-FP.11
  - Block 1 (set 2): 1-FP.15, 1.FP.18, 1.FP.20
  - Block 2 (set 3): 2-FP.4, 1-FP.10
- QuickCheck: read Appendix A-2 (under general)

This lecture: Higher-Order Functions

This lecture: Higher-Order Functions

- Function Composition

This lecture: Higher-Order Functions

- Function Composition
- Lambda abstraction

This lecture: Higher-Order Functions

- Function Composition
- Lambda abstraction
- Higher-Order Functions
  - `map`, `filter`, `zipWith`
  - `foldl`, `foldr`, `foldl1`, `foldr1`, etc.

This lecture: Higher-Order Functions

- Function Composition
- Lambda abstraction
- Higher-Order Functions
  - `map`, `filter`, `zipWith`
  - `foldl`, `foldr`, `foldl1`, `foldr1`, etc.
- (data structures)
  - If there is time...
  - Slides as reference

Functions

Recursive functions: maximum

```
maximum :: Ord a => [a] -> a
```

- What are the base cases?

Recursive functions: maximum

```
maximum :: Ord a => [a] -> a
```

- What are the base cases?

```
maximum [] = error "empty list"
```

Recursive functions: maximum

```
maximum :: Ord a => [a] -> a
```

- What are the base cases?

  ```
  maximum [] = error "empty list"
  ```

- Base case: `maximum` of a list with one element?

  ```
  maximum [x] = ...
  ```

Recursive functions: maximum

```
maximum :: Ord a => [a] -> a
```

- What are the base cases?

  ```
  maximum [] = error "empty list"
  ```

- Base case: `maximum` of a list with one element?

  ```
  maximum [x] = x
  ```

Recursive functions: maximum

```
maximum :: Ord a => [a] -> a
```

- What are the base cases?
  ```
  maximum [] = error "empty list"
  ```
- Base case: `maximum` of a list with one element?
  ```
  maximum [x] = x
  ```
- Recursive case: `maximum` of
  ```
  maximum (x:xs) = ...  maximum xs ...
  ```
  (assume that "`maximum xs`" already works for the shorter list)

Recursive functions: maximum

```
maximum :: Ord a => [a] -> a
```

- What are the base cases?
  ```
  maximum [] = error "empty list"
  ```
- Base case: `maximum` of a list with one element?
  ```
  maximum [x] = x
  ```
- Recursive case: `maximum` of
  ```
  maximum (x:xs) = max x $ maximum xs
  ```
  (assume that "`maximum xs`" already works for the shorter list)

Recursive functions: maximum

```
maximum :: Ord a => [a] -> a
```

- What are the base cases?
  ```
  maximum [] = error "empty list"
  ```
- Base case: `maximum` of a list with one element?
  ```
  maximum [x] = x
  ```
- Recursive case: `maximum` of
  ```
  maximum (x:xs) = max x $ maximum xs
  ```
  (assume that "`maximum xs`" already works for the shorter list)
- Recursive case using *guards*
  ```
  maximum (x:x':xs) | x > x'     = maximum (x  : xs)
                    | otherwise = maximum (x' : xs)
  ```

List comprehension

- Define a function `findI` that gives all indices of elements in a given list satisfying some predicate `p`

```
findI :: (a -> Bool) -> [a] -> [Int]
```

List comprehension

- Define a function `findI` that gives all indices of elements in a given list satisfying some predicate `p`

```
findI :: (a -> Bool) -> [a] -> [Int]
```

- Example:

```
findI even [12,3,1,99,100,101] == [0,4]
```

List comprehension

- Define a function `findI` that gives all indices of elements in a given list satisfying some predicate `p`

```
findI :: (a -> Bool) -> [a] -> [Int]
```

- Example:

```
findI even [12,3,1,99,100,101] == [0,4]
```

- Using *list comprehension*:

```
findI p xs = [  | ...]
```

List comprehension

- Define a function `findI` that gives all indices of elements in a given list satisfying some predicate `p`

```
findI :: (a -> Bool) -> [a] -> [Int]
```

- Example:

```
findI even [12,3,1,99,100,101] == [0,4]
```

- Using *list comprehension*:

```
findI p xs = [  | (i,x) <- zip [0..]  xs,...]
```

List comprehension

- Define a function `findI` that gives all indices of elements in a given list satisfying some predicate `p`

```
findI :: (a -> Bool) -> [a] -> [Int]
```

- Example:

```
findI even [12,3,1,99,100,101] == [0,4]
```

- Using *list comprehension*:

```
findI p xs = [  | (i,x) <- zip [0..]  xs,p x]
```

List comprehension

- Define a function `findI` that gives all indices of elements in a given list satisfying some predicate `p`

```
findI :: (a -> Bool) -> [a] -> [Int]
```

- Example:

```
findI even [12,3,1,99,100,101] == [0,4]
```

- Using *list comprehension*:

```
findI p xs = [ ...  | (i,x) <- zip [0..]  xs,p x]
```

List comprehension

- Define a function `findI` that gives all indices of elements in a given list satisfying some predicate `p`

```
findI :: (a -> Bool) -> [a] -> [Int]
```

- Example:

```
findI even [12,3,1,99,100,101] == [0,4]
```

- Using *list comprehension*:

```
findI p xs = [ i | (i,x) <- zip [0..]  xs,p x]
```

Quiz

Question 1

Consider the following function:

```
f :: y -> x -> y
f x y = x
```

Which of the following statements about this function is true?

A This Haskell code is accepted by the compiler

B This code does not compile since: brackets are missing, it should be: `f (x,y) = x`

C This code does not compile since: `y` is not used

D This code does not compile since: the type is incorrect and should be `f :: x -> y -> x`

Question 2

Consider the following function:

```
f (xs:x) = x
```

What is the result of the following expression?

```
f "abc"
```

A This is invalid Haskell code (not accepted by the compiler)

B `"bc"`

C `'c'`

D `"ab"`

Question 3

The function `max` has the type

```
max :: Ord a =>  a -> a -> a
```

What is the type of the following expression?

```
x = max 'a'
```

A `x :: Ord a => a -> a -> a`

B `x :: Ord a => a -> a`

C `x :: Char -> Char -> Char`

D `x :: Char -> Char`

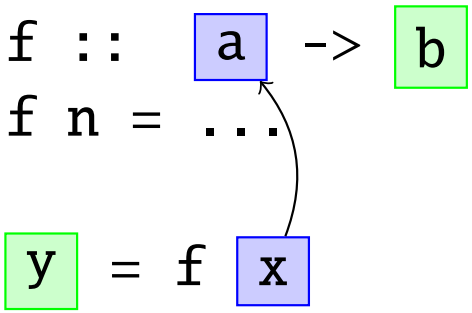
Function application

```
f ::   a -> b
f n = ...

y  = f  x
```

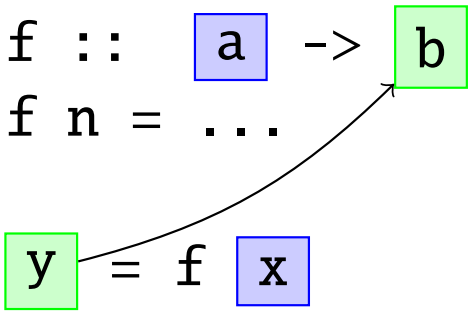
Function application

```
f ::  a -> b
f n = ...

y = f x
```

Function application

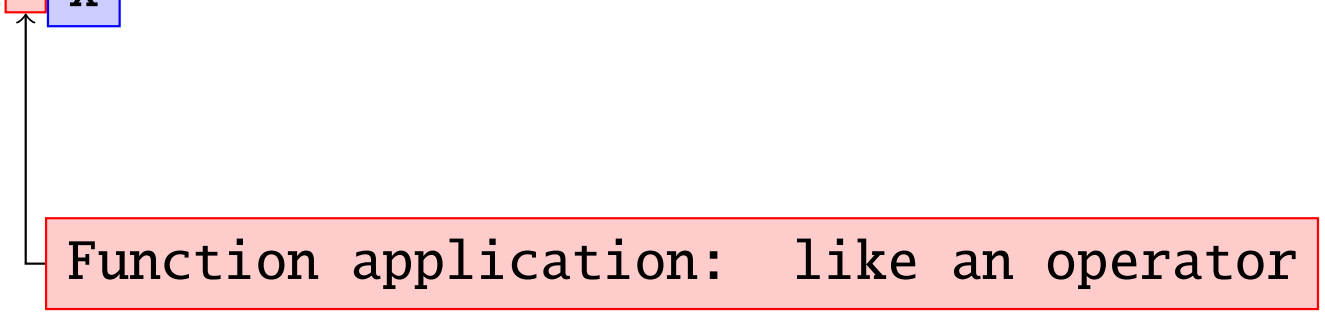


Function application



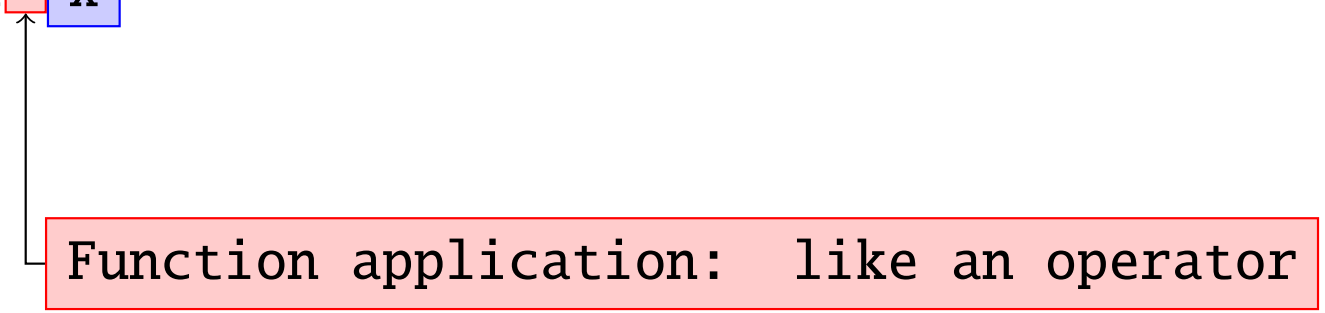
Function application

```
f ::  a -> b
f n = ...

y = f x
```



Function application

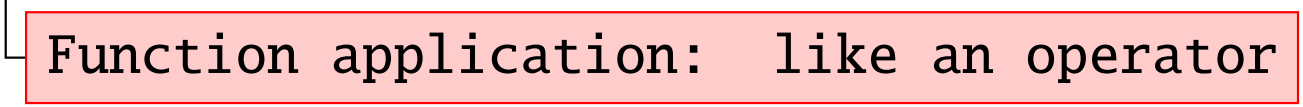
```
f ::  a -> b
f n = ...

y = f x
```



- Left associative: `f x + g y == (f x) + (g y)`

Function application

```
f ::  a -> b
f n = ...

y = f x
```

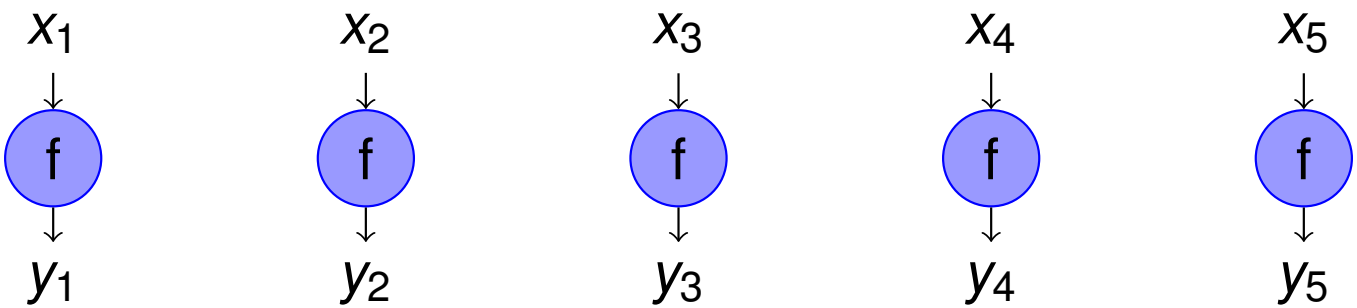


- Left associative: `f x + g y == (f x) + (g y)`
- `$` Right associative: `f $ x + g y == f (x + g y)`

```
($) :: (a -> b) -> a -> b
```

Higher-order functions: map

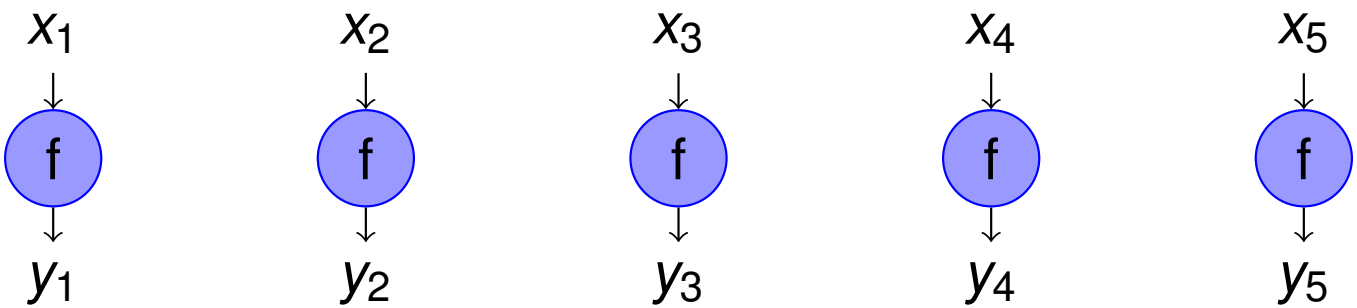
```
map :: (a -> b) -> [a] -> [b]
```




```
ys = map f xs
```

Higher-order functions: map

```
map :: (a -> b) -> [a] -> [b]
```

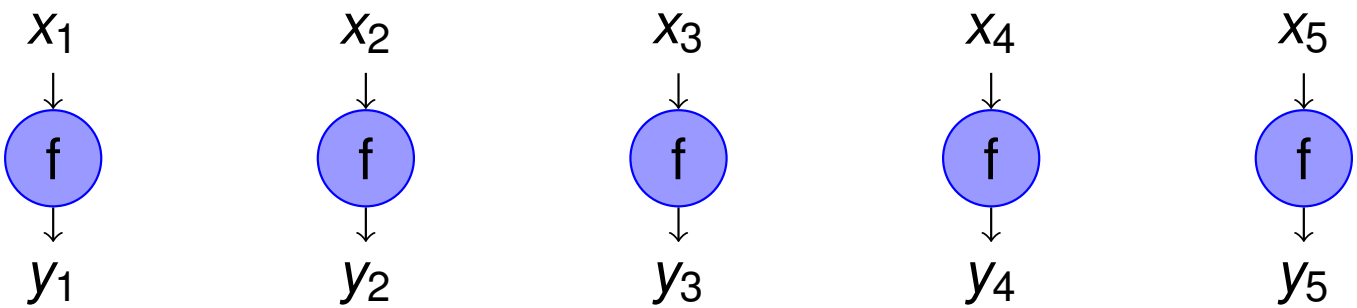



```
ys = map f xs
```

```
map (*2) [1,2,3] == [2,4,6]
```

Higher-order functions: map

```
map :: (a -> b) -> [a] -> [b]
```



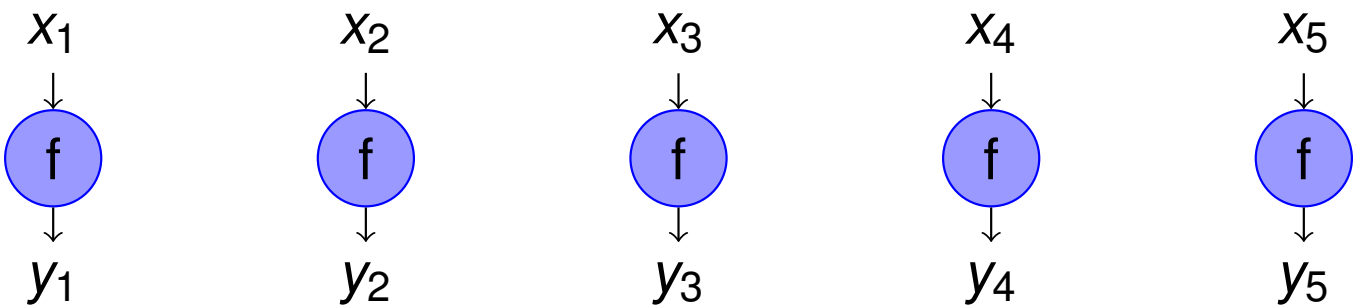

```
ys = map f xs
```

```
map (*2) [1,2,3] == [2,4,6]
```

```
map chr [72,101,108,108,111] == "Hello"
```

Higher-order functions: map

```
map :: (a -> b) -> [a] -> [b]
```

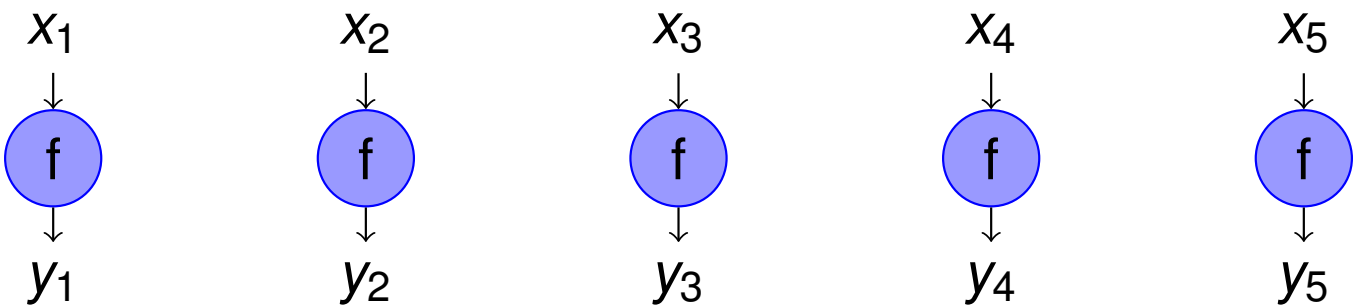



```
ys = map f xs
```

```
map (*2) [1,2,3] == [2,4,6]

map chr [72,101,108,108,111] == "Hello"

vlen vs = sqrt $ sum $ map (^2) vs
```

Higher-order functions: map

```
map :: (a -> b) -> [a] -> [b]
```




```
ys = map f xs
```

```
map (*2) [1,2,3] == [2,4,6]

map chr [72,101,108,108,111] == "Hello"

vlen vs = sqrt $ sum $ map (^2) vs

map reverse ["evil", "rats"] == ["live", "star"]
```

Currying

- Function of two arguments

  ```
  max :: Ord a => a -> a -> a
  ```

  Receives two arguments, it gives the maximum

Currying

- Function of two arguments

  ```
  max :: Ord a => a -> a -> a
  ```

  Receives two arguments, it gives the maximum

- Currying:

  ```
  max :: Ord a => a -> (a -> a)
  ```

  Receives *one* argument, it produces a new function (currying)

Currying

- Function of two arguments

  ```
  max :: Ord a => a -> a -> a
  ```

  Receives two arguments, it gives the maximum

- Currying:

  ```
  max :: Ord a => a -> (a -> a)
  ```

  Receives *one* argument, it produces a new function (currying)

- Example:

  ```
  max        :: Ord a             => a -> a -> a
  max 10     :: (Num a, Ord a) => a -> a
  max 10 20  :: (Num a, Ord a) => a
  ```

Currying

- Function of two arguments

```
max :: Ord a => a -> a -> a
```

  Receives two arguments, it gives the maximum

- Currying:

```
max :: Ord a => a -> (a -> a)
```

  Receives *one* argument, it produces a new function (currying)

- Example:

```
max       :: Ord a             => a -> a -> a
max 10    :: (Num a, Ord a) => a -> a
max 10 20 :: (Num a, Ord a) => a
```

- Correct Haskell code:

```
max 10 20 == (max 10) 20
```

Currying

- Function of two arguments

  ```
  max :: Ord a => a -> a -> a
  ```

  Receives two arguments, it gives the maximum

- Currying:

  ```
  max :: Ord a => a -> (a -> a)
  ```

  Receives *one* argument, it produces a new function (currying)

- Example:

  ```
  max        :: Ord a              => a -> a -> a
  max 10     :: (Num a, Ord a) => a -> a
  max 10 20  :: (Num a, Ord a) => a
  ```

- Correct Haskell code:

  ```
  max 10 20 == (max 10) 20
  ```

- **Partial application** with higher-order functions:

  ```
  map (max 0) [4, -2, 3, 0, -9, 5] == [4,0,3,0,0,5]
  ```

curry and uncurry

- (un)curry functions

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y     = f (x,y)

uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x,y) = f x y
```

curry and uncurry

- (un)curry functions

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y     = f (x,y)

uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x,y) = f x y
```

- Examples:

```
Prelude> let f (x,y) = x + y
Prelude> (curry f) 1 2
3
```

curry and uncurry

- (un)curry functions

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y     = f (x,y)

uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x,y) = f x y
```

- Examples:

```
Prelude> let f (x,y) = x + y
Prelude> (curry f) 1 2
3
Prelude> let g x y = x * y
Prelude> (uncurry g) (1, 2)
2
```

Lambda abstraction

- ```
  map f [1,2,3]
    where f x = 2*x + 1
  ```
  ```

Lambda abstraction

- ```
  map f [1,2,3]
    where f x = 2*x + 1
  ```

- Anonymous functions:

  ```
  f :: Num a => a -> a
  f x = 2*x + 1
  ```

  or:

  ```
  f :: Num a => a -> a
  f = \x -> 2*x + 1
  ```
  ```

Lambda abstraction

- ```
  map f [1,2,3]
    where f x = 2*x + 1
  ```

- Anonymous functions:

  ```
  f :: Num a => a -> a
  f x = 2*x + 1
  ```

  or:

  ```
  f :: Num a => a -> a
  f = \x -> 2*x + 1
  ```

- ```
  map (\x -> 2*x + 1) [1,2,3] == [3,5,7]
  ```

Lambda abstraction: curry

- Definition of `curry`:

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y     = f (x,y)
```

Lambda abstraction: curry

- Definition of `curry`:

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y      = f (x,y)
```

- Explicit alternative:

```
curry :: ((a, b) -> c) -> (a -> b -> c)
curry f          = \x y -> f (x,y)
```

Function composition (1/2)

Mathematics:

$$f(x) = g(h(x))$$

or:

$$f(x) = (g \circ h)(x)$$

or:

$$f = g \circ h$$

Function composition (1/2)

Mathematics:

$$f(x) = g(h(x))$$

or:

$$f(x) = (g \circ h)(x)$$

or:

$$f = g \circ h$$

Haskell:

```
f x = g (h x)
```

or:

```
f x = (g . h) x
```

or:

```
f = g . h
```

Function composition (2/2)

- Example:

```
odd = not . even
```

(pronounce as: `not` after `even`)

Function composition (2/2)

- Example:

```
odd = not . even
```

(pronounce as: `not` after `even`)

- Possible definition

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x ->  f (g x)
```

Function composition (2/2)

- Example:

  ```
  odd = not . even
  ```

  (pronounce as: `not` after `even`)

- Possible definition

  ```
  (.) :: (b -> c) -> (a -> b) -> a -> c
  f . g = \x ->  f (g x)
  ```

- Example of testing:

```
import Test.QuickCheck
import Test.QuickCheck.Function

prop_fc :: Fun Int Int -> Fun Int Int -> Int -> Bool
prop_fc (Fun _ f) (Fun _ g) x = f (g x) == (f . g) x
```

```
*Main> quickCheck prop_fc
+++ OK, passed 100 tests.
```

Higher order functions: filter

- `filter`: use a predicate function to filter elements from a list

```
filter :: (a -> Bool) -> [a] -> [a]
```

Higher order functions: filter

- `filter`: use a predicate function to filter elements from a list

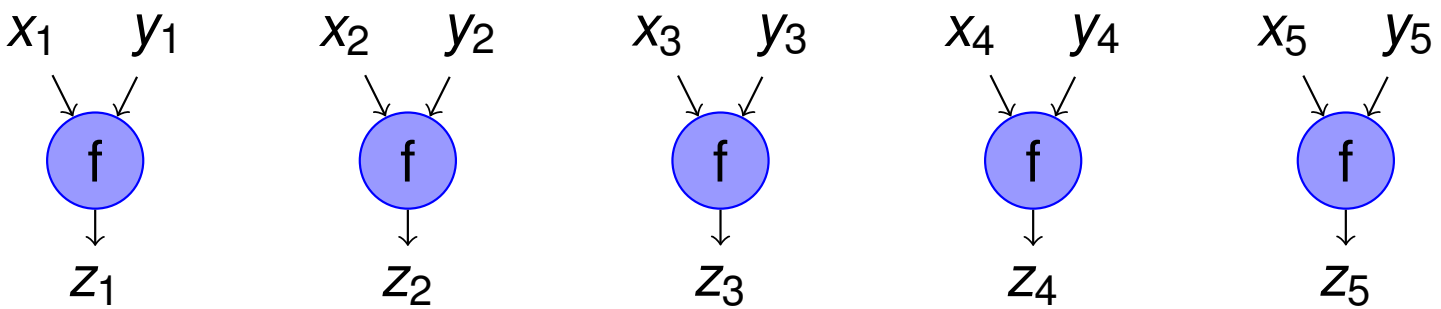
```
filter :: (a -> Bool) -> [a] -> [a]
```

- Examples:

```
filter (<10) [2,90,33,9,11,10] == [2, 9]

filter even [2,90,33,9,11,10] == [2,10]

findI p xs = map fst $ filter (p . snd) $ zip [0..] xs
```

Higher order functions: zipWith

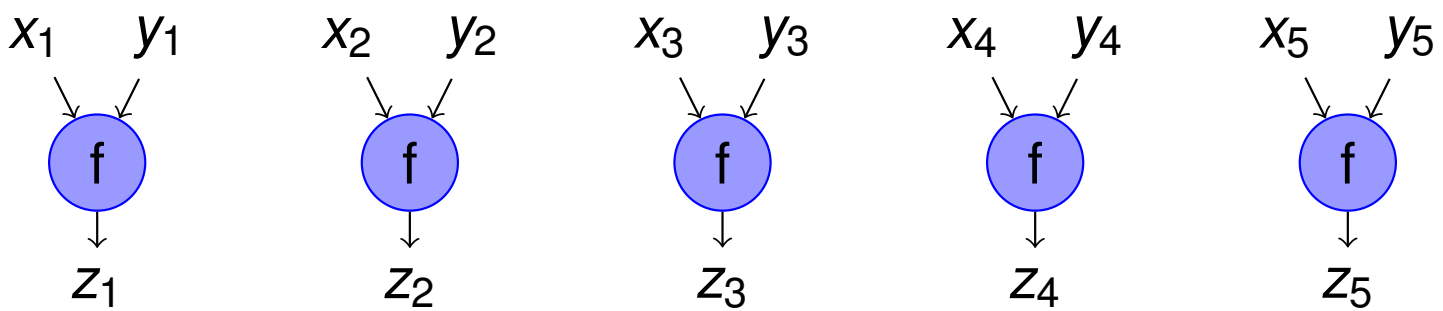
```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```




```
zs = zipWith f xs ys
```

Higher order functions: zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

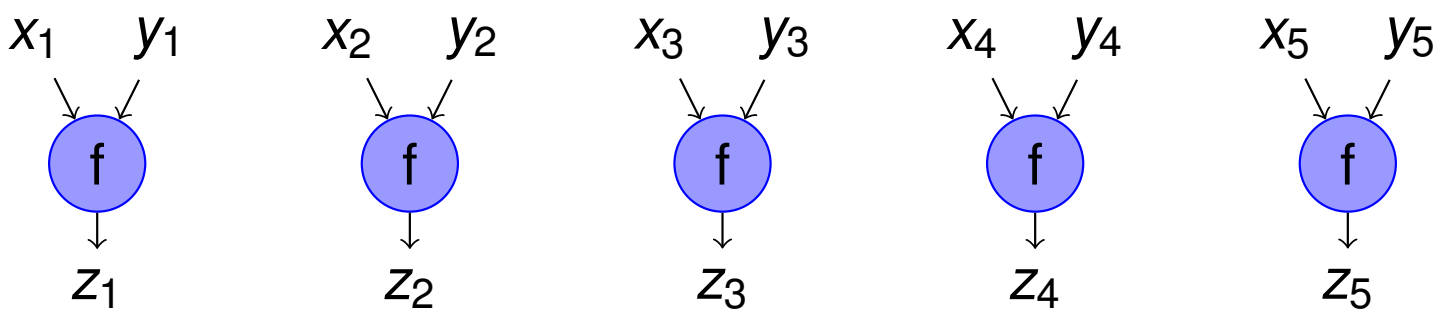



```
zs = zipWith f xs ys
```

```
zipWith (*) [1,2,3] [10,20,30] == [10,40,90]
```

Higher order functions: zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```



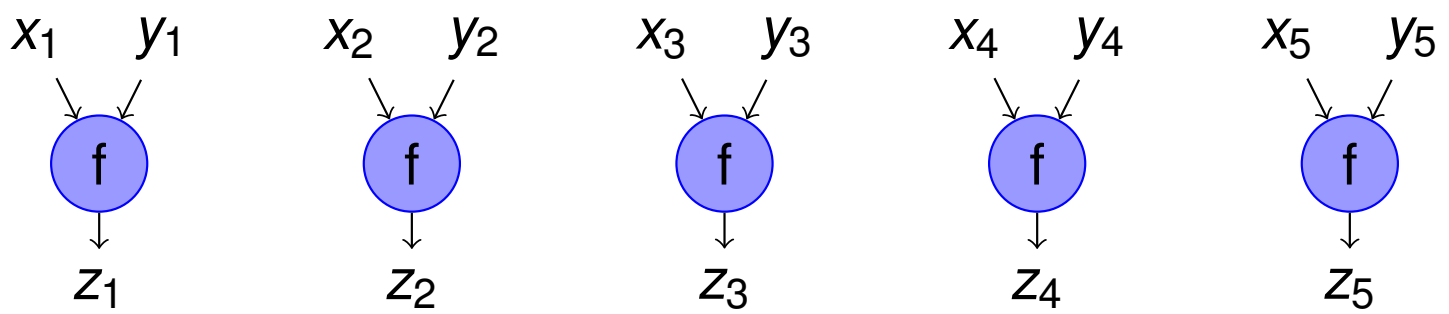

```
zs = zipWith f xs ys
```

```
zipWith (*) [1,2,3] [10,20,30] == [10,40,90]
```

```
zipWith max [100,2,3] [10,20,30] == [100,20,30]
```

Higher order functions: zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```




```
zs = zipWith f xs ys
```

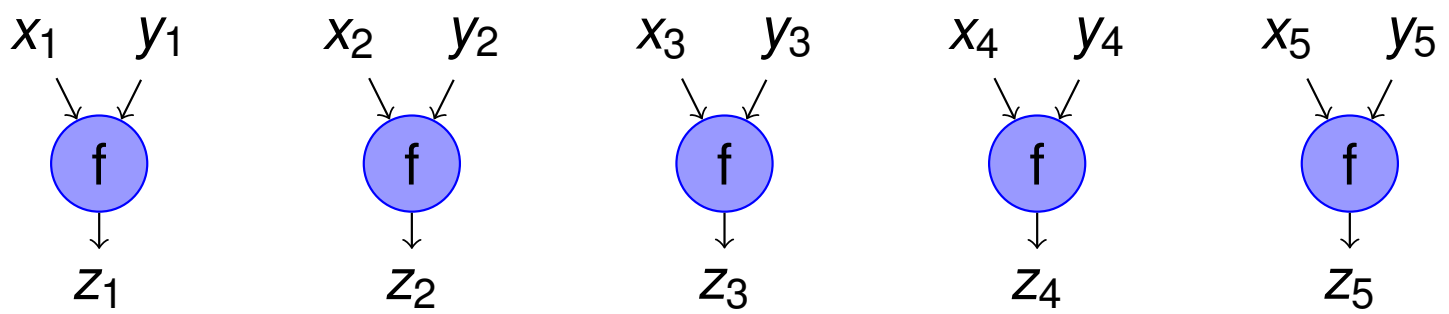
```
zipWith (*) [1,2,3] [10,20,30] == [10,40,90]
```

```
zipWith max [100,2,3] [10,20,30] == [100,20,30]
```

```
issorted xs = and $ zipWith (<=) xs (tail xs)
```

Higher order functions: zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```




```
zs = zipWith f xs ys
```

```
zipWith (*) [1,2,3] [10,20,30] == [10,40,90]

zipWith max [100,2,3] [10,20,30] == [100,20,30]

issorted xs = and $ zipWith (<=) xs (tail xs)

fibSeq = 0 : 1 : zipWith (+) fibSeq (tail fibSeq)
```

Higher order functions: Point Free Style

```
vlen :: [Double] -> Double
vlen vs = sqrt $ sum $ map (^2) vs
```

Higher order functions: Point Free Style

```
vlen :: [Double] -> Double
vlen vs = sqrt $ sum $ map (^2) vs
```

- The same as:

```
vlen :: [Double] -> Double
vlen vs = sqrt (sum (map (^2) vs))
```

Higher order functions: Point Free Style

```
vlen :: [Double] -> Double
vlen vs = sqrt $ sum $ map (^2) vs
```

- The same as:

  ```
  vlen :: [Double] -> Double
  vlen vs = sqrt (sum (map (^2) vs))
  ```

- First rewrite to function composition:

  ```
  vlen :: [Double] -> Double
  vlen vs = (sqrt . sum . map (^2)) vs
  ```

Higher order functions: Point Free Style

```
vlen :: [Double] -> Double
vlen vs = sqrt $ sum $ map (^2) vs
```

- The same as:

  ```
  vlen :: [Double] -> Double
  vlen vs = sqrt (sum (map (^2) vs))
  ```

- First rewrite to function composition:

  ```
  vlen :: [Double] -> Double
  vlen vs = (sqrt . sum . map (^2)) vs
  ```

- Point-free style: remove `vs`

  ```
  vlen :: [Double] -> Double
  vlen = sqrt . sum . map (^2)
  ```

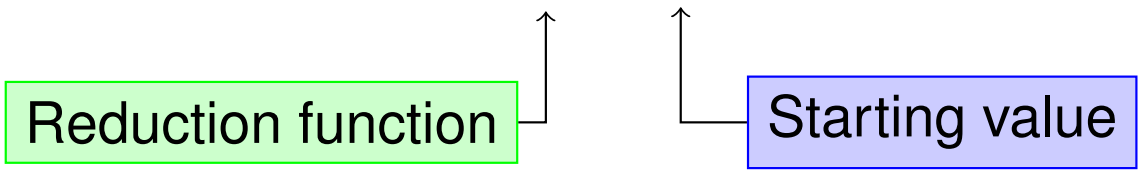
Higher order functions: Fold left

- Common patterns:
  - `sum [1,2,3,4] = 10`
  - `product [1,2,3,4] = 24`
  - `maximum [1,2,3,4] = 4`
  - `concat [[1,2,3], [4,5]] = [1,2,3,4,5]`

Higher order functions: Fold left

- Common patterns:
  - `sum [1,2,3,4] = 10`
  - `product [1,2,3,4] = 24`
  - `maximum [1,2,3,4] = 4`
  - `concat [[1,2,3], [4,5]] = [1,2,3,4,5]`
- `foldl :: (b -> a -> b) -> b -> [a] -> b`
- `foldl`: reduction of list:

```
sum xs = foldl (+) 0 xs
```



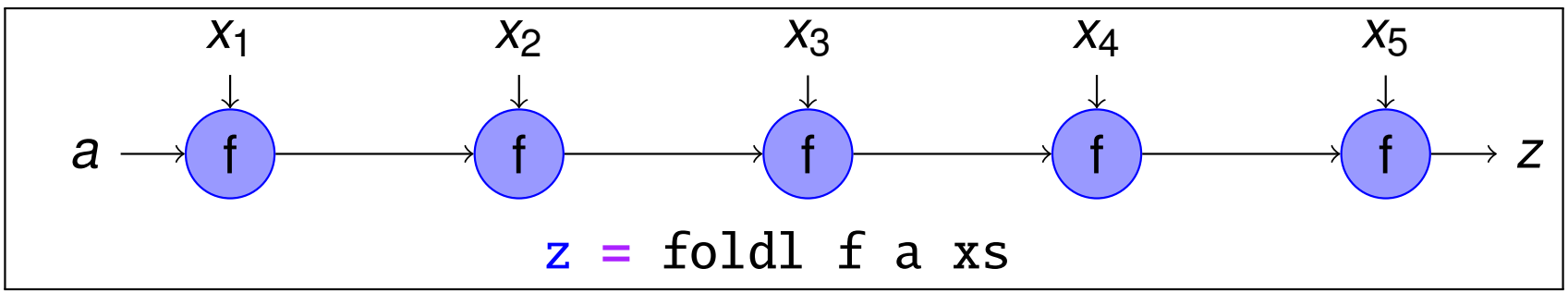
Higher order functions: Fold left

- Common patterns:
  - `sum [1,2,3,4] = 10`
  - `product [1,2,3,4] = 24`
  - `maximum [1,2,3,4] = 4`
  - `concat [[1,2,3], [4,5]] = [1,2,3,4,5]`
- `foldl :: (b -> a -> b) -> b -> [a] -> b`
- `foldl`: reduction of list:

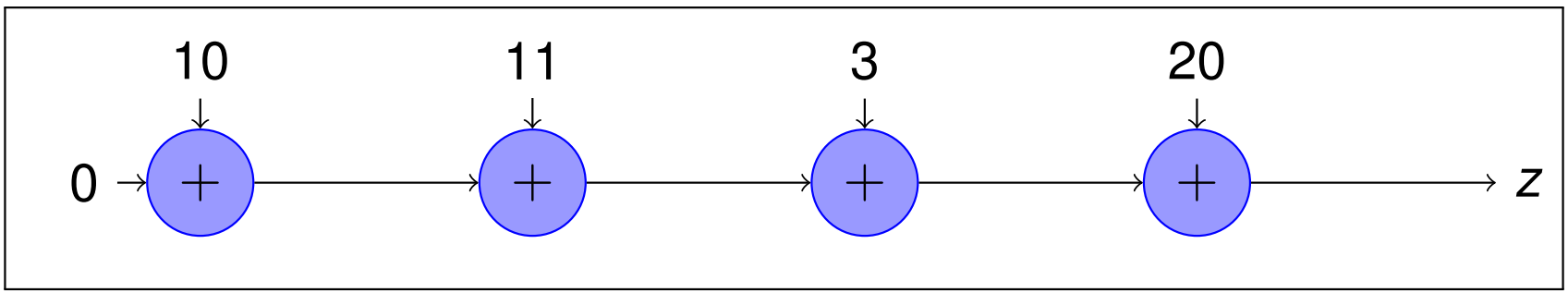
```
sum xs = foldl (+) 0 xs
```

Reduction function

Starting value

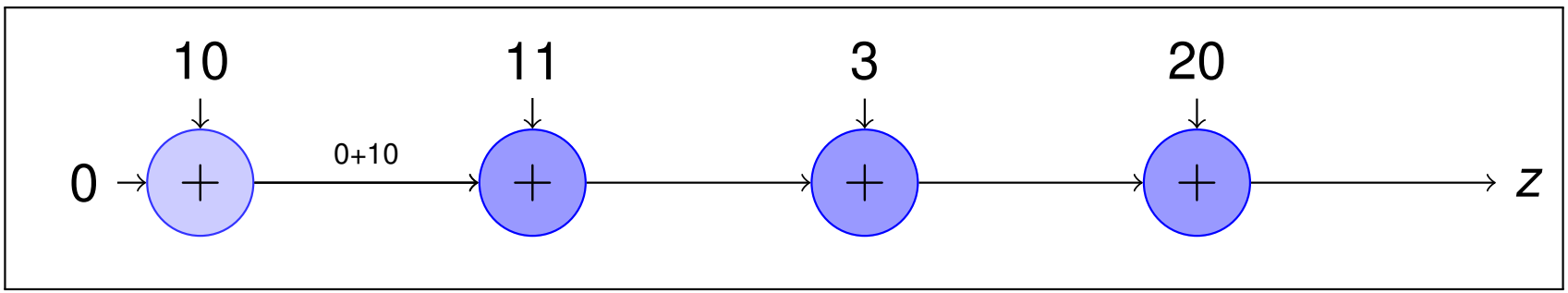


Fold left: Example



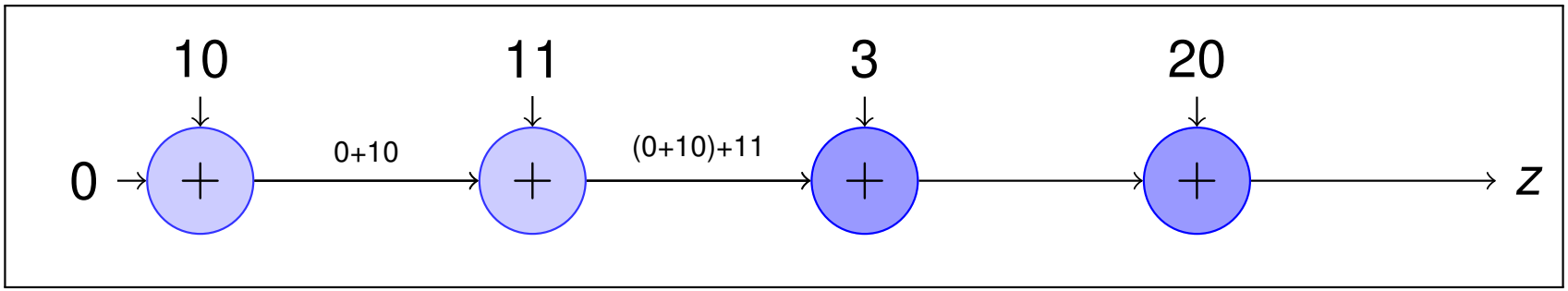

```
z  = foldl (+) 0                    [10,11,3,20]
```

Fold left: Example



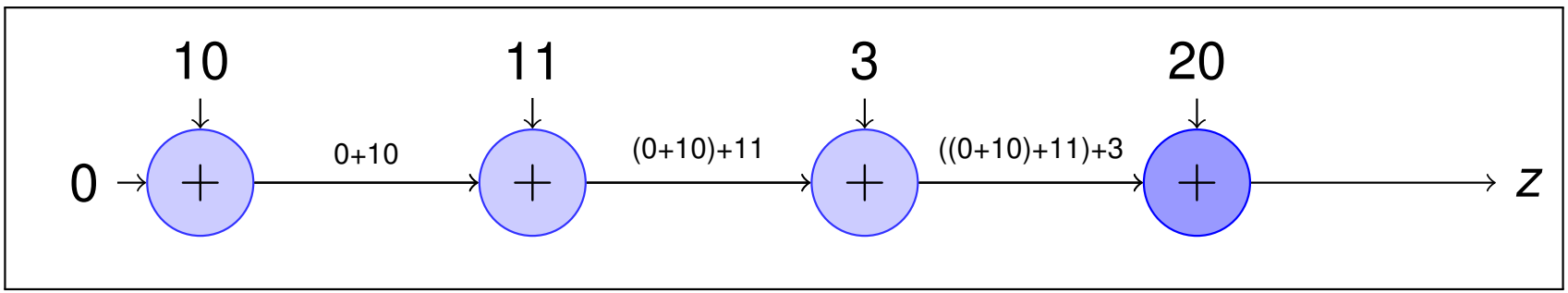

```
z  = foldl (+) 0                [10,11,3,20]
   = foldl (+) (0+10)           [11,3,20]
```

Fold left: Example



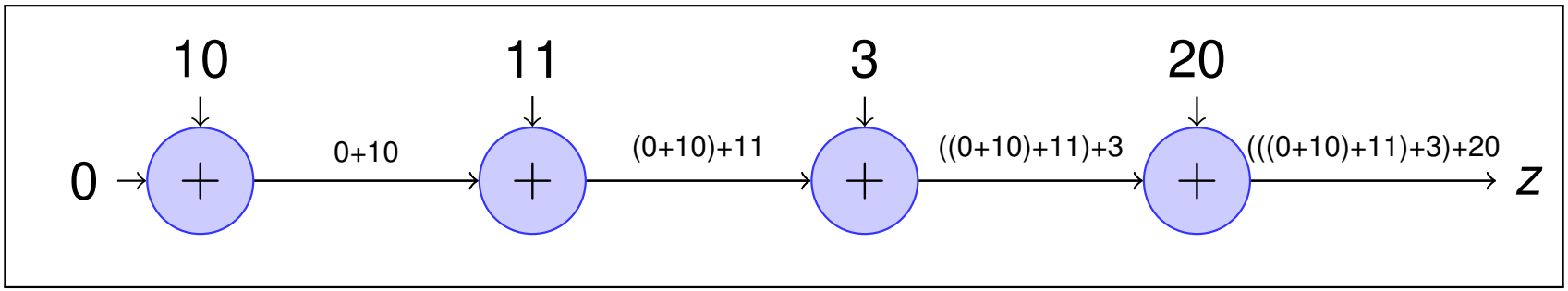

```
z  = foldl (+) 0                [10,11,3,20]
   = foldl (+) (0+10)           [11,3,20]
   = foldl (+) (0+10)+11)       [3,20]
```

Fold left: Example




```
z  = foldl (+) 0                    [10,11,3,20]
   = foldl (+) (0+10)                [11,3,20]
   = foldl (+) (0+10)+11)            [3,20]
   = foldl (+) ((0+10)+11)+3)        [20]
```

Fold left: Example




```
z  = foldl (+) 0                          [10,11,3,20]
   = foldl (+) (0+10)                     [11,3,20]
   = foldl (+) (0+10)+11)                 [3,20]
   = foldl (+) ((0+10)+11)+3)             [20]
   = foldl (+) (((0+10)+11)+3)+20)        []
   = ((0+10)+11)+3)+20
```

Other fold functions

- **Fold left:** left associative reduction

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl (#) x [x0,x1,..,xn] = (((x # x0) # x1)#..) # xn
```

Other fold functions

- **Fold left:** left associative reduction

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl (#) x [x0,x1,..,xn] = (((x # x0) # x1)#..) # xn
```

Other fold functions

- **Fold left:** left associative reduction

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl (#) x [x0,x1,..,xn] = (((x # x0) # x1)#..) # xn
```

Other fold functions

- **Fold left:** left associative reduction

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl (#) x [x0,x1,..,xn] = (((x # x0) # x1)#..) # xn
```

- **Fold right:** right associative reduction

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (#) x [x0,x1,..,xn] = x0 # (x1 # (.. (xn # x)))
```

Other fold functions

- **Fold left:** left associative reduction

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl (#) x [x0,x1,..,xn] = (((x # x0) # x1)#..) # xn
```

- **Fold right:** right associative reduction

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (#) x [x0,x1,..,xn] = x0 # (x1 # (.. (xn # x)))
```

- **foldl1:** use *first* element from list as starting value

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 (#) [x0,x1,..,xn] = ((x0 # x1) #..) # xn
```

Other fold functions

- **Fold left:** left associative reduction

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl (#) x [x0,x1,..,xn] = (((x # x0) # x1)#..) # xn
```

- **Fold right:** right associative reduction

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr (#) x [x0,x1,..,xn] = x0 # (x1 # (.. (xn # x)))
```

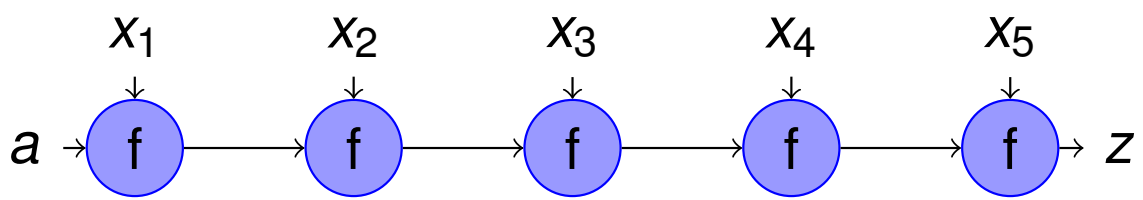
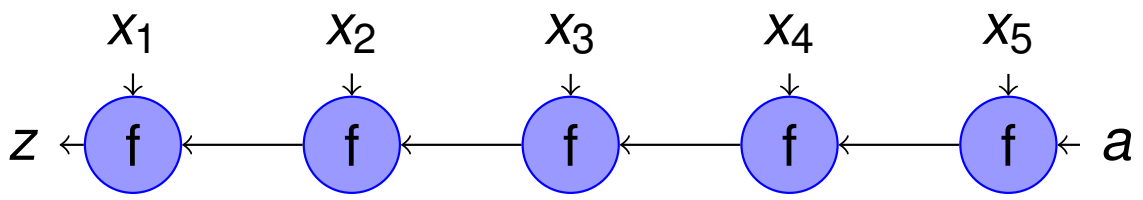
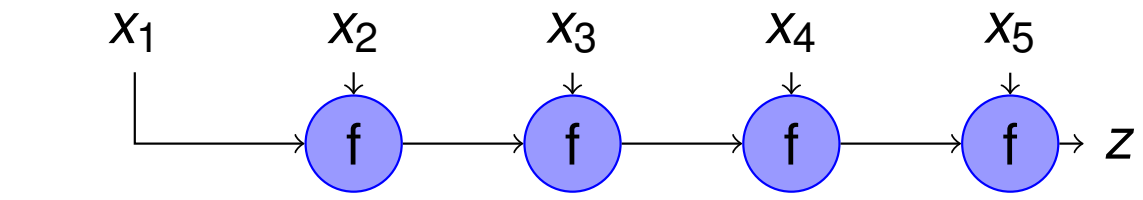
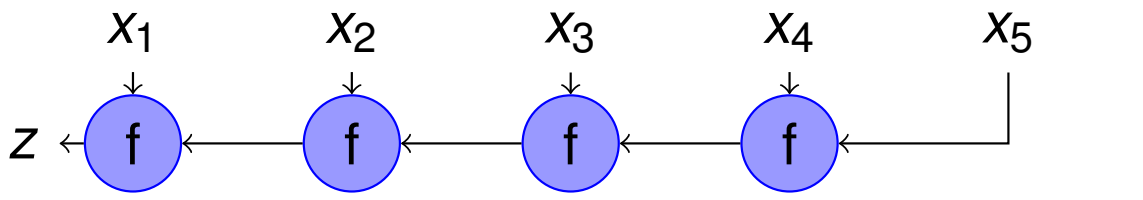
- **foldl1:** use *first* element from list as starting value

```
foldl1 :: (a -> a -> a) -> [a] -> a
foldl1 (#) [x0,x1,..,xn] = ((x0 # x1) #..) # xn
```

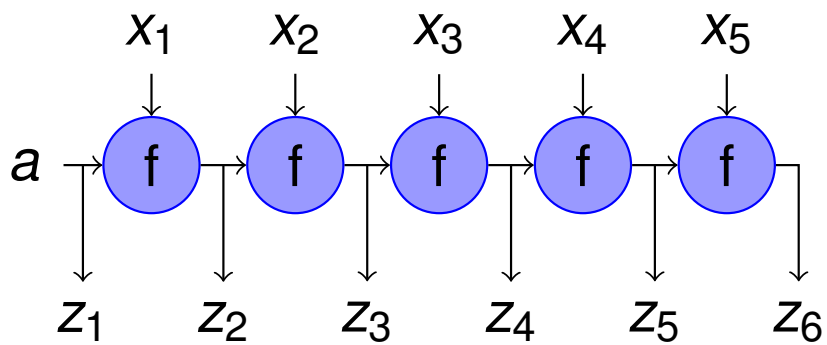
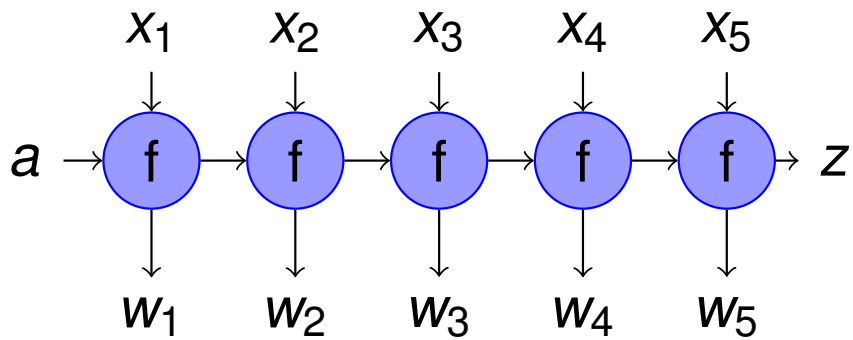
- **foldr1:** use *last* element from list as starting value

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 (#) [x0,x1,..,xn] = x0 # (x1 # (.. xn))
```

Fold functions: overview

`z = foldl f a xs`	
`z = foldr f a xs`	
`z = foldl1 f xs`	
`z = foldr1 f xs`	

Other higher order functions: overview

`zs = scanl f a xs`	
`(z,ws) = mapAccumL f a xs`	

Reminder: Compound types

- Lists: [a]

Reminder: Compound types

- Lists: `[a]`
- Tuples:
  - `()`
  - `(a,b)`
  - `(a,b,c)`
  - `(a,b,c,d)`
  - ...

Type synonyms

- Give another name to an *existing* type

```
type String = [Char]
```

Type synonyms

- Give another name to an *existing* type

```
type String = [Char]
```

- `String` and `[Char]` are now interchangeable:

```
firstname :: [Char]
firstname = "Ann"

lastname :: String
lastname = "Steward"

fullname = firstname ++ " " ++ lastname
```

Algebraic datatypes

```
data Bool = True | False
```

Algebraic datatypes

```
data Bool = True | False

not :: Bool -> Bool
not True  = False
not False = True
```

Algebraic datatypes

```
data Bool = True | False

not :: Bool -> Bool
not True  = False
not False = True

data Day = Monday
         | Tuesday
         | Wednesday
        ...
       deriving (Show, Ord)
```

Algebraic datatypes

```
data Bool = True | False

not :: Bool -> Bool
not True  = False
not False = True

data Day = Monday
         | Tuesday
         | Wednesday
        ...
       deriving (Show, Ord)

isweekend :: Day -> Bool
isweekend Saturday = True
isweekend Sunday   = True
isweekend _        = False
```

Value constructors

```haskell
data Shape = Rectangle Double Double -- width and height
           | Circle Double           -- radius
```

Value constructors

```
data Shape = Rectangle Double Double -- width and height
           | Circle Double           -- radius

unitcircle = Circle 1.0
square x   = Rectangle x x
```

Value constructors

```
data Shape = Rectangle Double Double -- width and height
           | Circle Double           -- radius

unitcircle = Circle 1.0
square x   = Rectangle x x

circumfence (Rectangle w h) = 2 * w + 2 * h
circumfence (Circle r)      = 2 * pi * r
```

Value constructors

```
data Shape = Rectangle Double Double -- width and height
           | Circle Double           -- radius

unitcircle = Circle 1.0
square x   = Rectangle x x

circumfence (Rectangle w h) = 2 * w + 2 * h
circumfence (Circle r)      = 2 * pi * r

area (Rectangle w h) = w * h
area (Circle r)      = pi * r^2
```

Recursive data types

```
data IntList = Nil                  -- similar: []
             | Cons Int IntList     -- similar: (:) x xs
```

Recursive data types

```haskell
data IntList = Nil                    -- similar: []
             | Cons Int IntList       -- similar: (:) x xs


testlist :: IntList
testlist  = Cons 1 ( Cons 2 ( Cons 3   Nil))
testlist' = Cons 1 $ Cons 2 $ Cons 3 $ Nil
```

Recursive data types

```haskell
data IntList = Nil                    -- similar: []
             | Cons Int IntList       -- similar: (:) x xs


testlist :: IntList
testlist  = Cons 1 ( Cons 2 ( Cons 3   Nil))
testlist' = Cons 1 $ Cons 2 $ Cons 3 $ Nil


total Nil          = 0
total (Cons x xs) = x + total xs
```

Recursive data types

```haskell
data IntList = Nil                    -- similar: []
             | Cons Int IntList       -- similar: (:) x xs


testlist :: IntList
testlist  = Cons 1 ( Cons 2 ( Cons 3   Nil))
testlist' = Cons 1 $ Cons 2 $ Cons 3 $ Nil


total Nil          = 0
total (Cons x xs) = x + total xs


total' Nil          = 0
total' (x `Cons` xs) = x + total xs
```

Self study

- This lecture: “Higher-order functions”
  - Learn you a Haskell (Lipovaca): Chapters 6
  - Programming in Haskell (Hutton): Chapters 7

Self study

- This lecture: “Higher-order functions”
  - Learn you a Haskell (Lipovaca): Chapters 6
  - Programming in Haskell (Hutton): Chapters 7
- Lecture Monday (29-04):
  - Additional lecture; no new material
  - Quiz; work in pairs; bring paper - no laptops!
  - Other view on challenging subjects
  - Please provide input!

Self study

- This lecture: “Higher-order functions”
  - Learn you a Haskell (Lipovaca): Chapters 6
  - Programming in Haskell (Hutton): Chapters 7
- Lecture Monday (29-04):
  - Additional lecture; no new material
  - Quiz; work in pairs; bring paper - no laptops!
  - Other view on challenging subjects
  - Please provide input!
- Lecture Wednesday (01-05): “Types and Type Classes”
  - Learn you a Haskell (Lipovaca): Chapter 8
  - Programming in Haskell (Hutton): Chapter 8, Chapter 12.1