# Programming in Haskell
# Aug-Nov 2015

## LECTURE 8

## AUGUST 27, 2015

S P Suresh
Chennai Mathematical Institute

# Combining elements

# Combining elements

```
sumlist :: [Int] -> Int
sumlist [] = 0
sumlist (x:xs) = x + (sumlist xs)
```

# Combining elements

```
sumlist :: [Int] -> Int
sumlist [] = 0
sumlist (x:xs) = x + (sumlist xs)

multlist :: [Int] -> Int
multlist [] = 1
multlist (x:xs) = x * (multlist xs)
```

# Combining elements

```
sumlist :: [Int] -> Int
sumlist [] = 0
sumlist (x:xs) = x + (sumlist xs)

multlist :: [Int] -> Int
multlist [] = 1
multlist (x:xs) = x * (multlist xs)
```

- What is the common pattern?

# Combining elements …

# Combining elements ...

```
combine f v [] = v
combine f v (x:xs) = f x (combine f v xs)
```

# Combining elements ...

```
combine f v [] = v
combine f v (x:xs) = f x (combine f v xs)
```

- We can then write

# Combining elements ...

```
combine f v [] = v
combine f v (x:xs) = f x (combine f v xs)
```

- We can then write

```
sumlist l  = combine (+) 0 l
```

# Combining elements ...

```
combine f v [] = v
combine f v (x:xs) = f x (combine f v xs)
```

- We can then write

```
sumlist l  = combine (+) 0 l

multlist l = combine (*) 1 l
```

# foldr

- The built-in version of `combine` is called `foldr`

```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

# foldr

- The built-in version of `combine` is called `foldr`

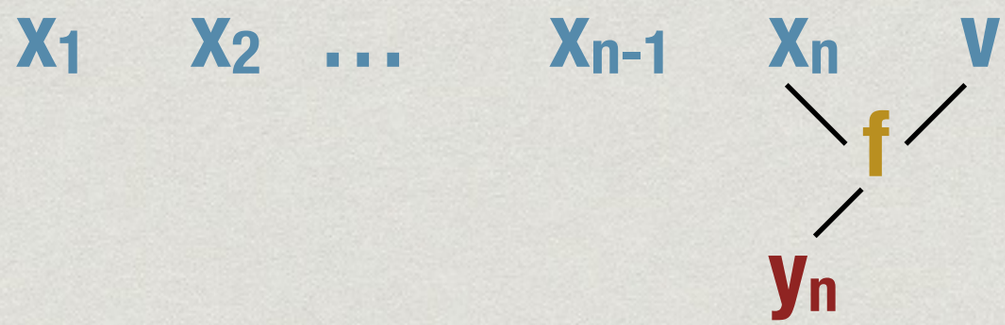
```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

$x_1 \quad x_2 \quad \ldots \quad x_{n-1} \quad x_n \quad v$

# foldr

- The built-in version of `combine` is called `foldr`

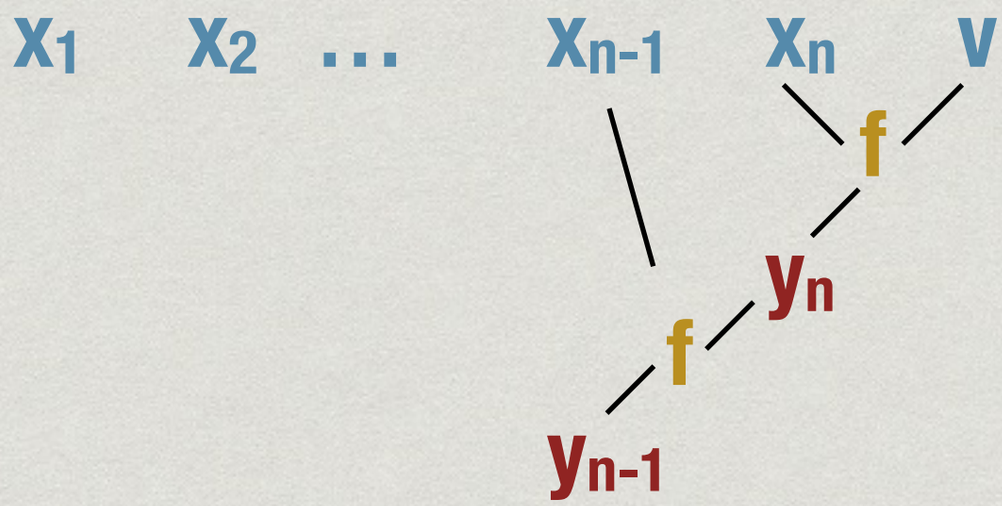
```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```



# foldr

- The built-in version of `combine` is called `foldr`

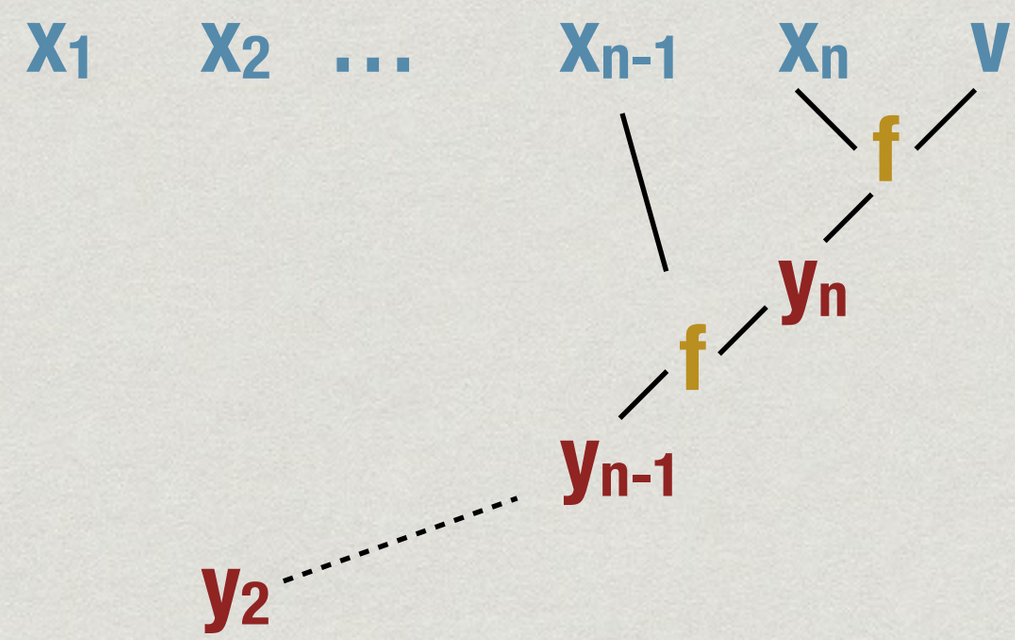
```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```



# foldr

- The built-in version of `combine` is called `foldr`

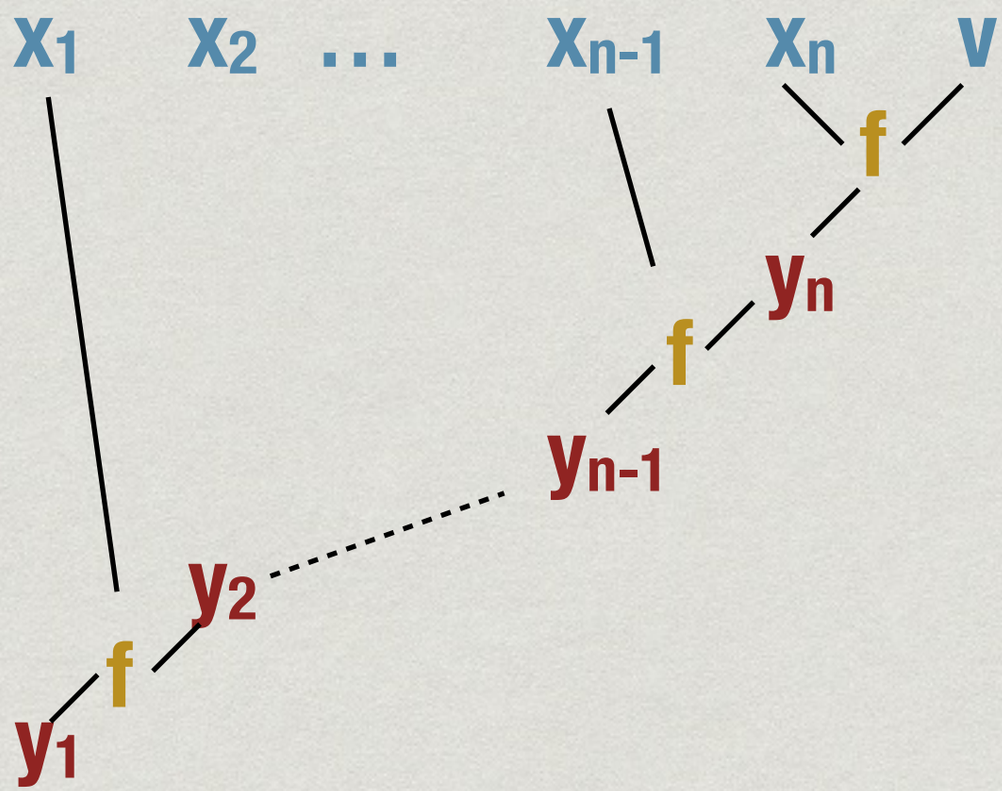
```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```



# foldr

- The built-in version of `combine` is called `foldr`

```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```



# Examples

# Examples

* `sumlist l = foldr (+) 0 l`

# Examples

- `sumlist l = foldr (+) 0 l`
- `multlist l = foldr (*) 1 l`

# Examples

* `sumlist l = foldr (+) 0 l`
* `multlist l = foldr (*) 1 l`
* 
```
mylength :: [Int] -> Int
mylength l =  foldr f 0 l
  where
  f x y = y+1
```

# Examples

- `sumlist l = foldr (+) 0 l`
- `multlist l = foldr (*) 1 l`
- 
```
mylength :: [Int] -> Int
mylength l =  foldr f 0 l
  where
  f x y = y+1
```

- Note: can simply write `mylength = foldr f 0`

# Examples

- `sumlist l = foldr (+) 0 l`
- `multlist l = foldr (*) 1 l`
- 
```
mylength :: [Int] -> Int
mylength l =  foldr f 0 l
  where
  f x y = y+1
```

- Note: can simply write `mylength = foldr f 0`
  - Outermost reduction: `mylength l ⇛ foldr f 0 l`

# Examples ...

# Examples ...

- Recall

# Examples ...

- Recall

```
appendright x l = l ++ [x]
```

# Examples ...

- Recall

```
appendright x l = l ++ [x]
```

- `foldr appendright [] = ??`

# Examples ...

- Recall

```
appendright x l = l ++ [x]
```

- `foldr appendright [] = ??`
- `foldr appendright [] = reverse`

# Examples ...

# Examples ...

- What is `foldr (++) []`?

# Examples ...

- What is `foldr (++) []`?
- Dissolves one level of brackets

# Examples ...

- What is `foldr (++) []`?

- Dissolves one level of brackets

  - Flattens a list of lists into a single list

# Examples ...

- What is `foldr (++) []`?

- Dissolves one level of brackets

  - Flattens a list of lists into a single list

- The built-in function `concat`

# foldr

# foldr

```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

# foldr

```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

- What is the type of `foldr`?

# foldr

```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

- What is the type of `foldr`?

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

# foldr

```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

- What is the type of `foldr`?

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

# foldr1

# foldr1

- Sometimes there is no natural value to assign to the empty list

# foldr1

- Sometimes there is no natural value to assign to the empty list
- Finding the maximum value in the list

# foldr1

- Sometimes there is no natural value to assign to the empty list
- Finding the maximum value in the list
  - Maximum is undefined for empty list

# foldr1

- Sometimes there is no natural value to assign to the empty list
- Finding the maximum value in the list
  - Maximum is undefined for empty list

```
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

# foldr1

- Sometimes there is no natural value to assign to the empty list
- Finding the maximum value in the list
  - Maximum is undefined for empty list

```
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)

maxlist = foldr1 max
```

# Folding from the left

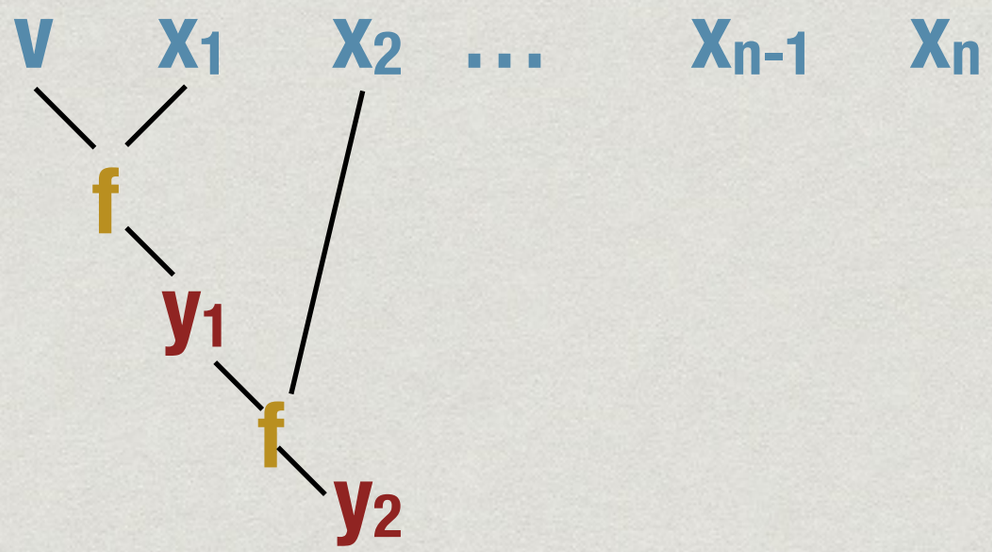
- Sometimes useful to fold left to right

- ```
  foldl :: (a -> b -> a) -> a -> [b] -> a
  foldl f v [] = v
  foldl f v (x:xs) = foldl f (f v x) xs
  ```
  ```

# Folding from the left

- Sometimes useful to fold left to right

- ```
  foldl :: (a -> b -> a) -> a -> [b] -> a
  foldl f v [] = v
  foldl f v (x:xs) = foldl f (f v x) xs
  ```

$$v \quad x_1 \quad x_2 \quad \ldots \quad x_{n-1} \quad x_n$$
  ```

# Folding from the left

- Sometimes useful to fold left to right

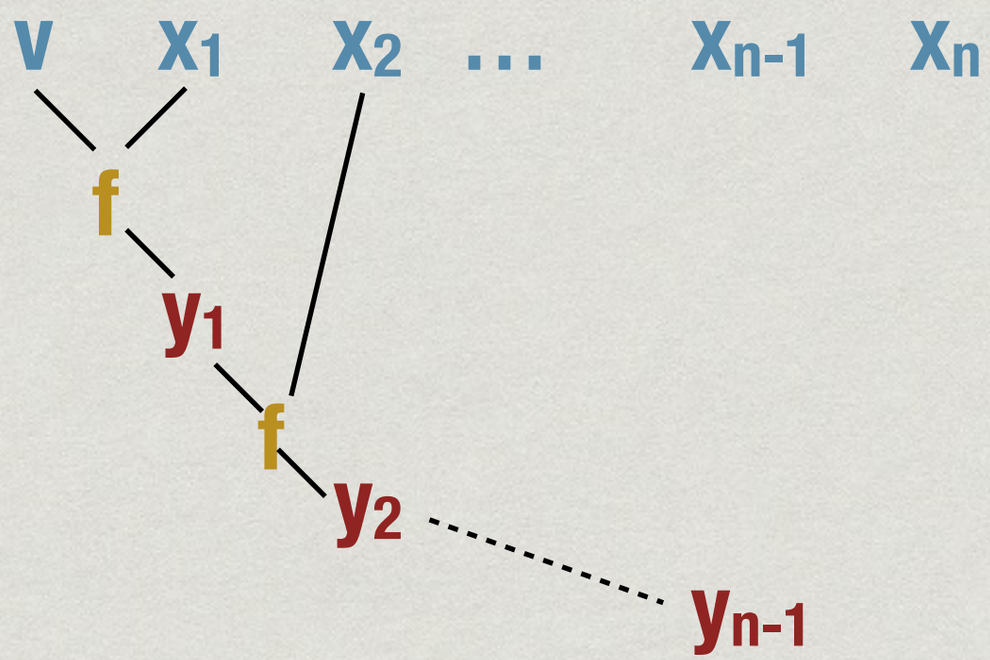
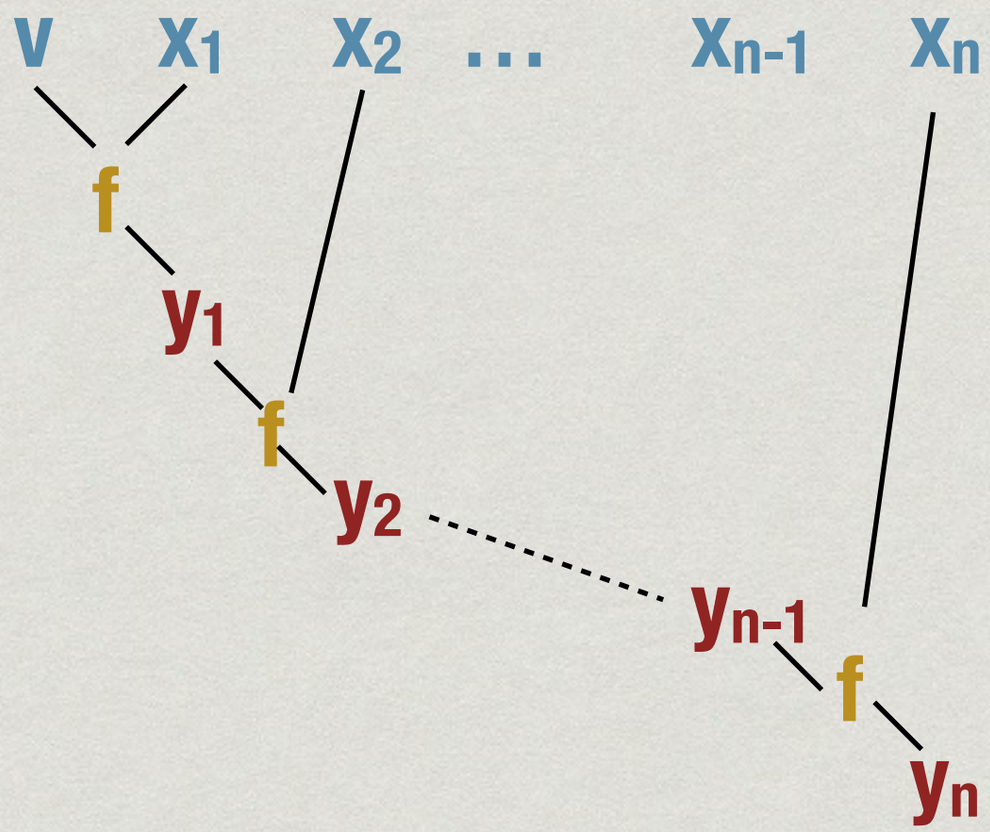
- ```
  foldl :: (a -> b -> a) -> a -> [b] -> a
  foldl f v [] = v
  foldl f v (x:xs) = foldl f (f v x) xs
  ```


  ```

# Folding from the left

- Sometimes useful to fold left to right

- ```
  foldl :: (a -> b -> a) -> a -> [b] -> a
  foldl f v [] = v
  foldl f v (x:xs) = foldl f (f v x) xs
  ```


  ```

# Folding from the left

- Sometimes useful to fold left to right

- ```
  foldl :: (a -> b -> a) -> a -> [b] -> a
  foldl f v [] = v
  foldl f v (x:xs) = foldl f (f v x) xs
  ```


  ```

# Folding from the left

- Sometimes useful to fold left to right

- ```
  foldl :: (a -> b -> a) -> a -> [b] -> a
  foldl f v [] = v
  foldl f v (x:xs) = foldl f (f v x) xs
  ```


  ```

# Example

# Example

- Translate a string of digits to an integer

# Example

- Translate a string of digits to an integer

```
strtonum "234" = 234
```

# Example

- Translate a string of digits to an integer

```
strtonum "234" = 234
```

- Convert a character into the corresponding digit:

# Example

- Translate a string of digits to an integer

```
strtonum "234" = 234
```

- Convert a character into the corresponding digit:

```
chartonum :: Char -> Int
chartonum c
    | ('0' <= c) && (c <= '9')
                  = (ord c) - (ord '0')
```

# Example …

- Process the digits left to right
- Multiply current sum by 10 and add next digit

```
nextdigit :: Int -> Char -> Int
nextdigit i c = 10*i + (chartonum c)

strtonum = foldl nextdigit 0
```

# takeWhile

# takeWhile

- `take n l` returns `n` element prefix of list `l`

# takeWhile

- `take n l` returns `n` element prefix of list `l`
- Instead, use a property to determine the prefix

# takeWhile

- `take n l` returns `n` element prefix of list `l`
- Instead, use a property to determine the prefix
- `takeWhile :: (a -> Bool) -> [a] -> [a]`

# takeWhile

- `take n l` returns `n` element prefix of list `l`
- Instead, use a property to determine the prefix
- `takeWhile :: (a -> Bool) -> [a] -> [a]`
- `takeWhile (> 7) [8,1,9,10] = [8]`

# takeWhile

- `take n l` returns `n` element prefix of list `l`
- Instead, use a property to determine the prefix
- `takeWhile :: (a -> Bool) -> [a] -> [a]`
- `takeWhile (> 7) [8,1,9,10] = [8]`
- `takeWhile (< 10) [8,1,9,10]= [8,1,9]`

# Example: `position`

# Example: `position`

- `position c s` : first position in `s` where `c` occurs

# Example: `position`

- `position c s` : first position in `s` where `c` occurs

```
position :: Char -> String -> Int
position c "" = 0
position c (d:ds)
    | c == d    = 0
    | otherwise = 1 + (position c ds)
```

# Example: `position`

- `position c s` : first position in `s` where `c` occurs

```
position :: Char -> String -> Int
position c "" = 0
position c (d:ds)
    | c == d    = 0
    | otherwise = 1 + (position c ds)
```

- Using `takeWhile`

# Example: `position`

- `position c s` : first position in `s` where `c` occurs

```
position :: Char -> String -> Int
position c "" = 0
position c (d:ds)
    | c == d    = 0
    | otherwise = 1 + (position c ds)
```

- Using `takeWhile`

```
position c s = length (takeWhile (/= c) s)
```

# Example: `position`

- `position c s` : first position in `s` where `c` occurs

```
position :: Char -> String -> Int
position c "" = 0
position c (d:ds)
    | c == d    = 0
    | otherwise = 1 + (position c ds)
```

- Using `takeWhile`

```
position c s = length (takeWhile (/= c) s)
```

- Symmetric function `dropWhile`