Programming in Haskell
Aug-Nov 2015

LECTURE 10

SEPTEMBER 3, 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`

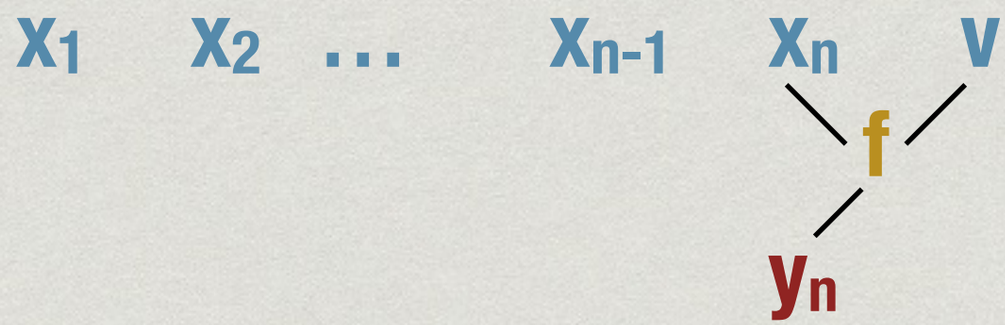
```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

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

foldr

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

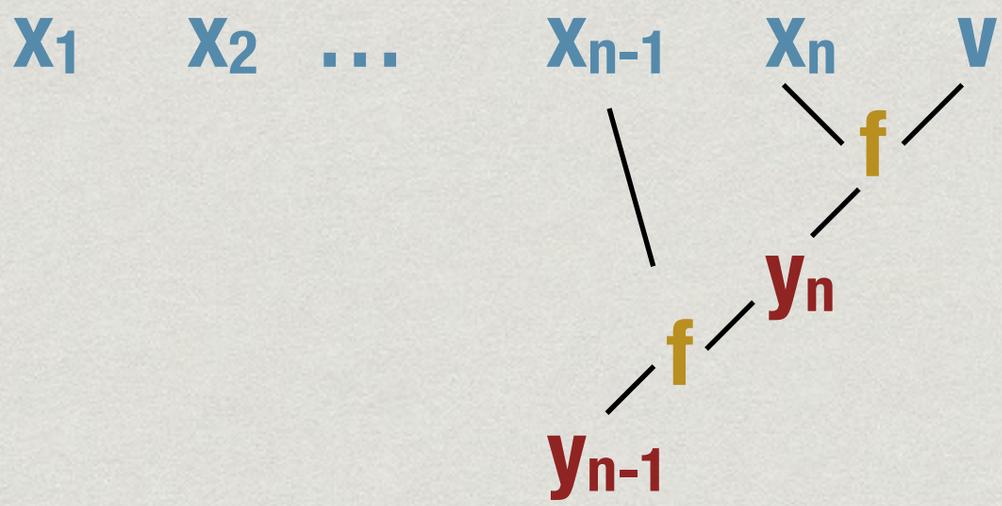
```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```



foldr

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

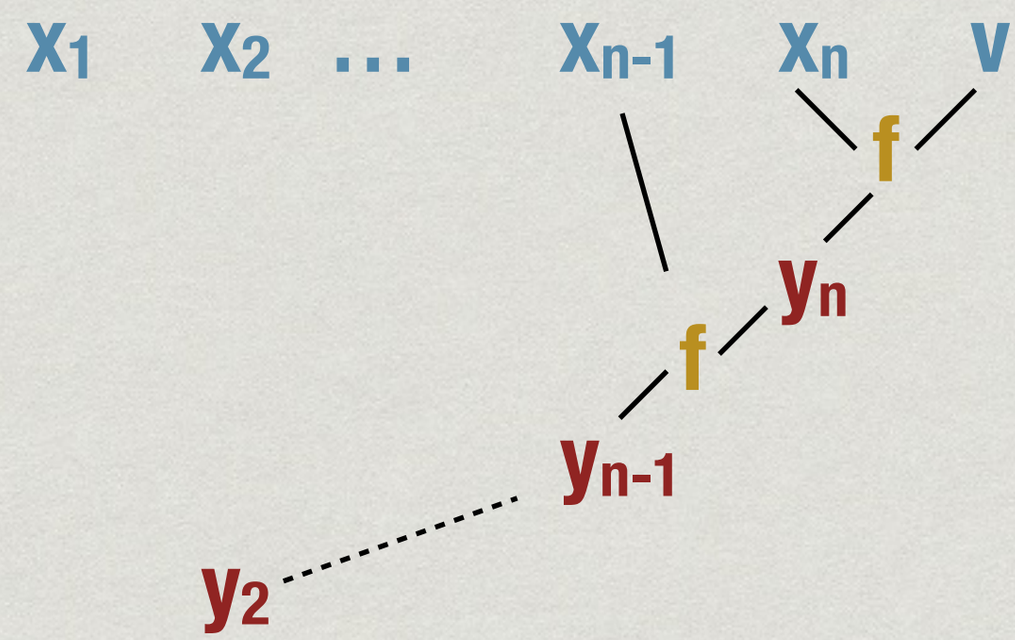
```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```



foldr

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

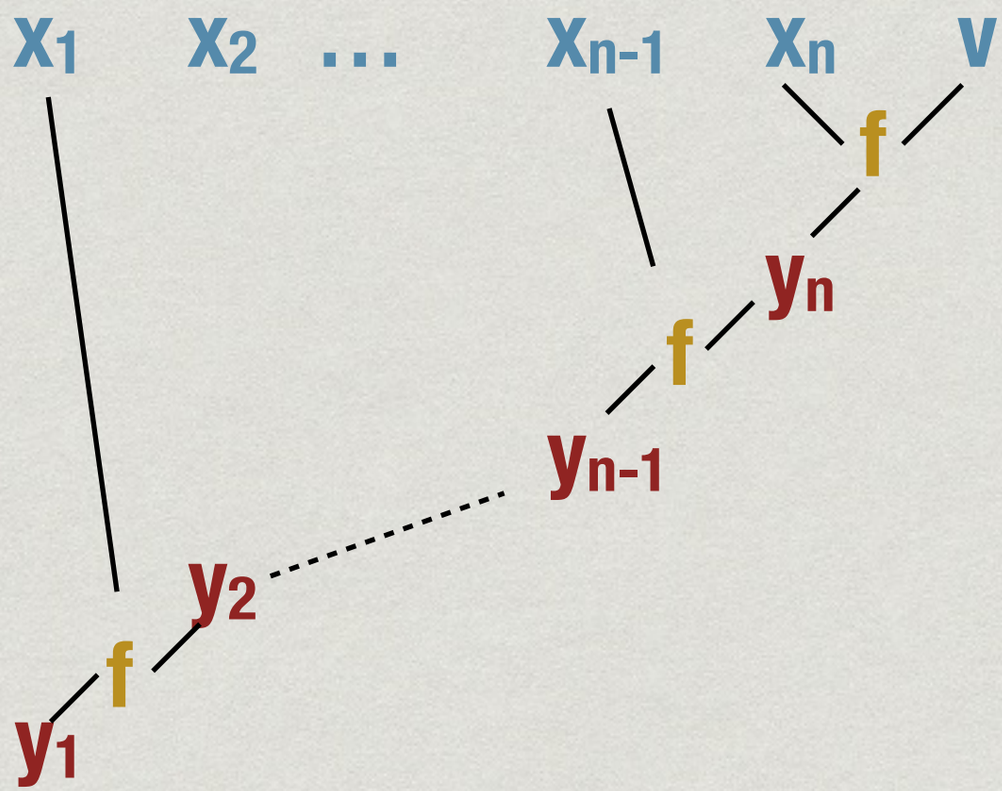
```
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```



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

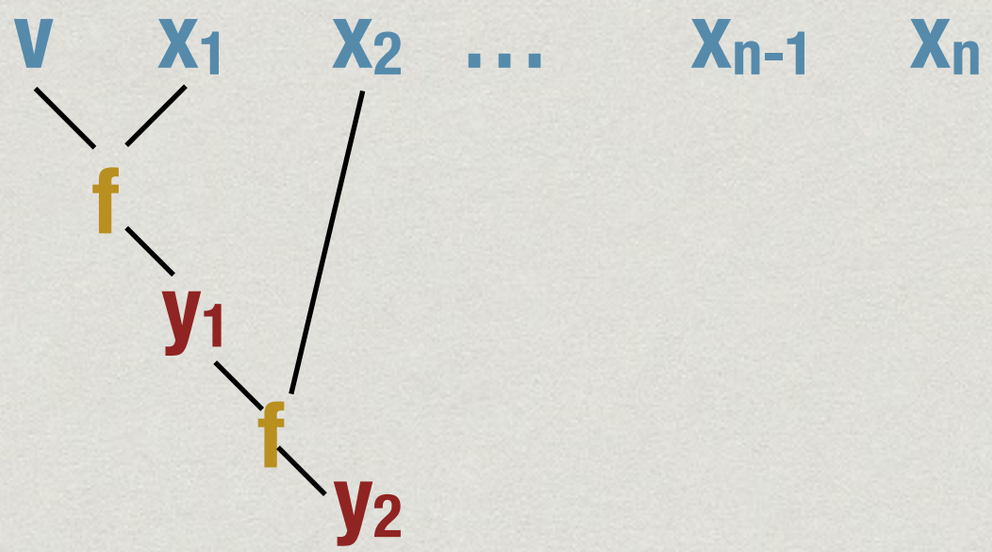
- Sometimes useful to fold left to right

- ```
  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

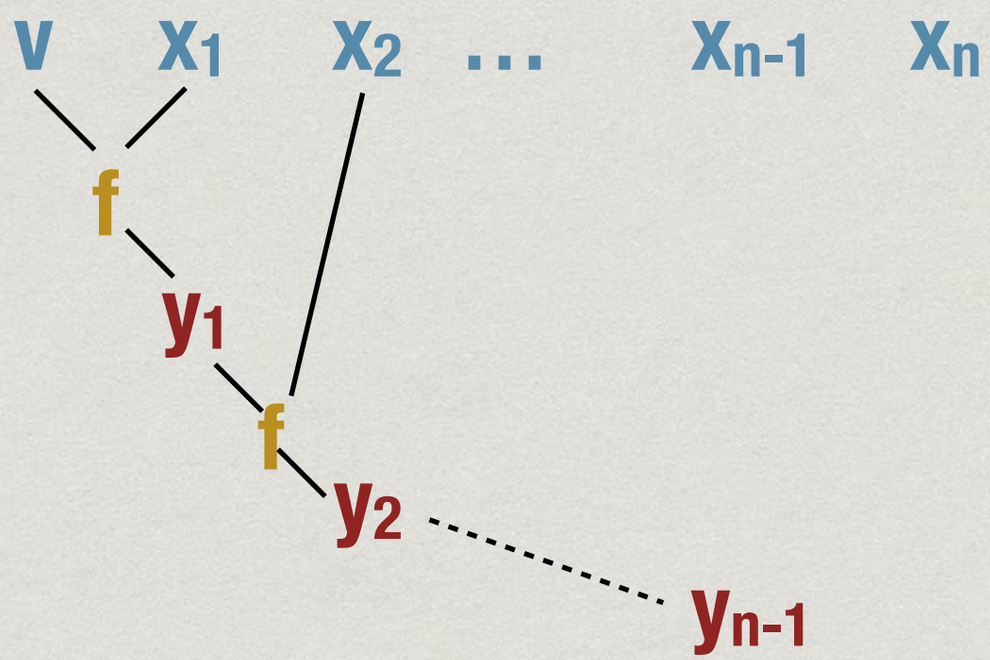
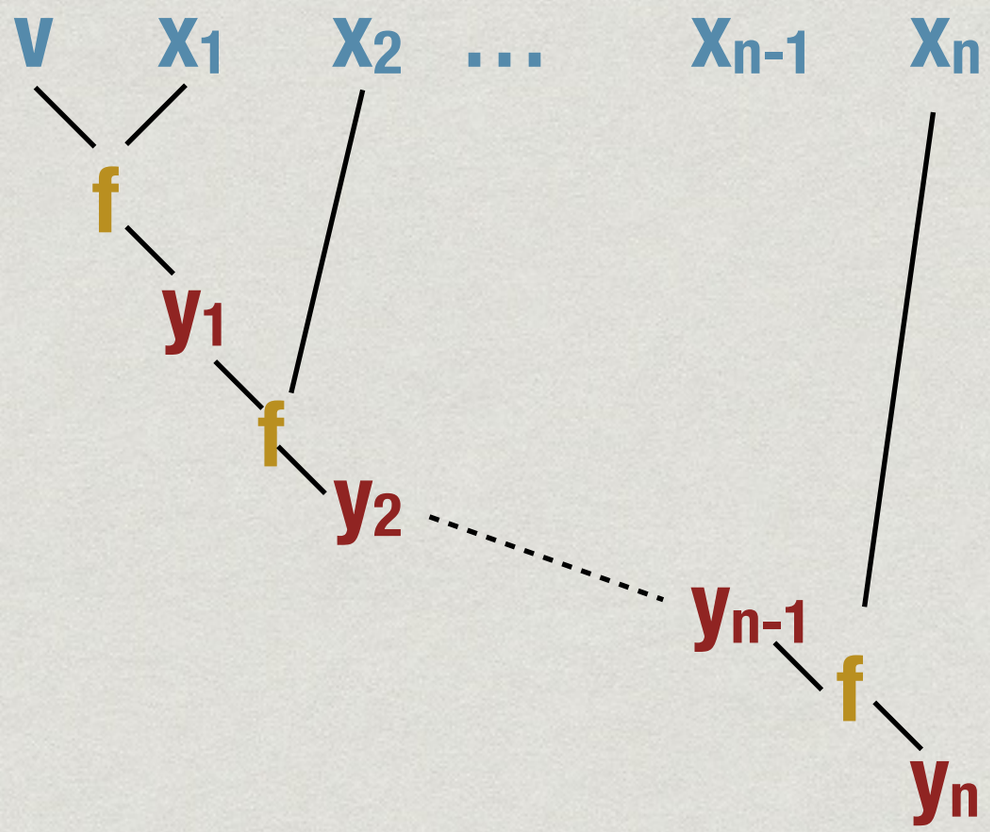
- ```
  foldl :: (a -> b -> a) -> a -> [b] -> a
  foldl f v [] = v
  foldl f v (x:xs) = foldl f (f v x) xs
  ```


  ```

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`

zipWith

zipWith

- `map f l` applies `f` to every element of `l`

zipWith

- `map f l` applies `f` to every element of `l`
- `zipWith` combines two lists using a function

zipWith

- `map f l` applies `f` to every element of `l`
- `zipWith` combines two lists using a function
  - `zipWith (+) [1,2,3] [7,8,6] = [8,10,9]`

zipWith

- `map f l` applies `f` to every element of `l`
- `zipWith` combines two lists using a function
  - `zipWith (+) [1,2,3] [7,8,6] = [8,10,9]`
  - `zipWith (+) [1,2,3] [7,8] = [8,10]`

zipWith

- `map f l` applies `f` to every element of `l`
- `zipWith` combines two lists using a function
  - `zipWith (+) [1,2,3] [7,8,6] = [8,10,9]`
  - `zipWith (+) [1,2,3] [7,8] = [8,10]`
  - `zipWith (<) [1,2] [2,1] = [True,False]`

zipWith

- `map f l` applies `f` to every element of `l`
- `zipWith` combines two lists using a function
  - `zipWith (+) [1,2,3] [7,8,6] = [8,10,9]`
  - `zipWith (+) [1,2,3] [7,8] = [8,10]`
  - `zipWith (<) [1,2] [2,1] = [True,False]`
  - `zipWith :: (a -> b -> c)->[a]->[b]->[c]`

Example: Mark lists

Example: Mark lists

* `marks :: [[Int]]`

Example: Mark lists

- `marks :: [[Int]]`
  - Each list in `marks` corresponds to an assignment.

Example: Mark lists

- `marks :: [[Int]]`

- Each list in `marks` corresponds to an assignment.

- Entry j is the mark obtained by student j

Example: Mark lists

- `marks :: [[Int]]`

- Each list in `marks` corresponds to an assignment.
- Entry j is the mark obtained by student j
- Compute the total marks obtained by each student.

Example: Mark lists

Example: Mark lists

- `addMarks [[10,10,8], [9,2,10], [8,2,8]] = [27,14,26]`

Example: Mark lists

- `addMarks [[10,10,8], [9,2,10], [8,2,8]] = [27,14,26]`
- `addMarks [[3,4],[2]] = [5]`

Example: Mark lists

* `addMarks [[10,10,8], [9,2,10], [8,2,8]] = [27,14,26]`
* `addMarks [[3,4],[2]] = [5]`

```
addmarks :: [[Int]] -> [Int]
addMarks [x] = x
addMarks (x:xs) = zipWith (+) x (addMarks xs)
```

Example: Mark lists

- `addMarks [[10,10,8], [9,2,10], [8,2,8]] = [27,14,26]`
- `addMarks [[3,4],[2]] = [5]`

```
addmarks :: [[Int]] -> [Int]
addMarks [x] = x
addMarks (x:xs) = zipWith (+) x (addMarks xs)

addMarks = foldr1 (zipWith (+))
```

zip

zip

- Combine two lists into a list of pairs

zip

- Combine two lists into a list of pairs
- `zip :: [a] -> [b] -> [(a,b)]`

zip

- Combine two lists into a list of pairs
- `zip :: [a] -> [b] -> [(a,b)]`
- `zip` stops with the shorter of the two lists

zip

- Combine two lists into a list of pairs
- `zip :: [a] -> [b] -> [(a,b)]`
- `zip` stops with the shorter of the two lists
- `zip ['a','b','c'] [1..3]`
  ⇛ `[('a',1),('b',2),('c',3)]`

zip

- Combine two lists into a list of pairs
- `zip :: [a] -> [b] -> [(a,b)]`
- `zip` stops with the shorter of the two lists
- `zip ['a','b','c'] [1..3]`
  `⇛ [('a',1),('b',2),('c',3)]`
- `zip ['a'..'z'] [1..10]`
  `⇛ [('a',1),('b',2),...,('j',10)]`

Example

Example

- Check that a list of integers is non-decreasing

Example

- Check that a list of integers is non-decreasing

- ```
  nondecreasing :: [Int] -> Bool
  nondecreasing [] = True
  nondecreasing [x] = True
  nondecreasing (x:y:ys) =
                        (x <= y) && (nondecreasing (y:ys))
  ```
  ```

Example

- Check that a list of integers is non-decreasing

- ```
  nondecreasing :: [Int] -> Bool
  nondecreasing [] = True
  nondecreasing [x] = True
  nondecreasing (x:y:ys) =
                        (x <= y) && (nondecreasing (y:ys))
  ```

- We are checking `x1 <= x2 && x2 <= x3 && …`
  ```

Example

- Check that a list of integers is non-decreasing

- ```
  nondecreasing :: [Int] -> Bool
  nondecreasing [] = True
  nondecreasing [x] = True
  nondecreasing (x:y:ys) =
                        (x <= y) && (nondecreasing (y:ys))
  ```

- We are checking `x1 <= x2 && x2 <= x3 && …`

- Pair up `l` with `tail l` and compare position by position
  ```

Example

- Check that a list of integers is non-decreasing

- ```
  nondecreasing :: [Int] -> Bool
  nondecreasing [] = True
  nondecreasing [x] = True
  nondecreasing (x:y:ys) =
                      (x <= y) && (nondecreasing (y:ys))
  ```

- We are checking `x1 <= x2 && x2 <= x3 && …`

- Pair up `l` with `tail l` and compare position by position

- ```
  nondecreasing xs =
              and [x <= y | (x,y) <- zip xs (tail xs) ]
  ```

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
- Tag each element in `s` by its position:

Example: `position`

- `position c s` : first position in `s` where `c` occurs
- Tag each element in `s` by its position:

```
zip s [0..(length s)]
```

Example: `position`

- `position c s` : first position in `s` where `c` occurs
- Tag each element in `s` by its position:

```
zip s [0..(length s)]
```

- Find all indices where `c` occurs

Example: `position`

- `position c s` : first position in `s` where `c` occurs
- Tag each element in `s` by its position:

```
zip s [0..(length s)]
```

- Find all indices where `c` occurs

```
allpos c s =
    [i | (y,i) <- zip s [0..(length s)], c == y]
```

Example: `position`

- `position c s` : first position in `s` where `c` occurs
- Tag each element in `s` by its position:

```
zip s [0..(length s)]
```

- Find all indices where `c` occurs

```
allpos c s =
    [i | (y,i) <- zip s [0..(length s)], c == y]
```

- Append “not found” value and extract first element

Example: `position`

- `position c s` : first position in `s` where `c` occurs
- Tag each element in `s` by its position:

```
zip s [0..(length s)]
```

- Find all indices where `c` occurs

```
allpos c s =
    [i | (y,i) <- zip s [0..(length s)], c == y]
```

- Append "not found" value and extract first element

```
position c s = head ((allpos c s) ++ [length s])
```

Example: initial segments

Example: initial segments

- Write a Haskell function `initsegs` which returns the list of initial segments of a list.

Example: initial segments

- Write a Haskell function `initsegs` which returns the list of initial segments of a list.

  - `initsegs [1,2,3] = [[],[1],[1,2],[1,2,3]]`

Example: initial segments

- Write a Haskell function `initsegs` which returns the list of initial segments of a list.

  - `initsegs [1,2,3] = [[],[1],[1,2],[1,2,3]]`

```
initsegs :: [a] -> [[a]]
initsegs [] = [[]]
initsegs (x:xs) =
                  []:map (x:) (initsegs xs)
```

Example: interleave

Example: interleave

- Insert `x` into list `l` at all possible positions and return the resulting set of lists

Example: interleave

- Insert `x` into list `l` at all possible positions and return the resulting set of lists

- `interleave 1 [2,3] =`
  `[[1,2,3],[2,1,3],[2,3,1]]`

Example: interleave

- Insert `x` into list `l` at all possible positions and return the resulting set of lists
- `interleave 1 [2,3] =`
  `[[1,2,3],[2,1,3],[2,3,1]]`

```
interleave :: a -> [a] -> [[a]]
interleave x [] = [[x]]
interleave x (y:ys) =
    (x:y:ys) : map (y:) (interleave x ys)
```

Example: Permutations

Example: Permutations

- Write down function that computes all the permutations of a given list (in any order)

Example: Permutations

- Write down function that computes all the permutations of a given list (in any order)

- `perm [1,2,3] =`
  `[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]`

Example: Permutations

- Write down function that computes all the permutations of a given list (in any order)

- `perm [1,2,3] =`
  `[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]`

```
perm :: [a] -> [[a]]
perm [x] = [[x]]
perm (x:xs) = concat (map (interleave x) (perm xs))
```

Example: Permutations

- Write down function that computes all the permutations of a given list (in any order)

- `perm [1,2,3] =`
  `[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]`

```
perm :: [a] -> [[a]]
perm [x] = [[x]]
perm (x:xs) = concat (map (interleave x) (perm xs))
```

- Built-in function:

Example: Permutations

- Write down function that computes all the permutations of a given list (in any order)

- `perm [1,2,3] =`
  `[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]`

```
perm :: [a] -> [[a]]
perm [x] = [[x]]
perm (x:xs) = concat (map (interleave x) (perm xs))
```

- Built-in function:

```
concatMap f l = concat (map f l)
```

Example: Partitions

Example: Partitions

- Compute all partitions of a list `l`

Example: Partitions

- Compute all partitions of a list `l`
- Partition of `l` : collection of nonempty lists `l1`, `l2`, ..., `lk` such that `l == l1 ++ l2 ++ ... ++ lk`

Example: Partitions

- Compute all partitions of a list `l`
- Partition of `l` : collection of nonempty lists `l1`, `l2`, ..., `lk` such that `l == l1 ++ l2 ++ ... ++ lk`
  - `part [1,2,3] = [[[1],[2],[3]],[[1,2],[3]],[[1],[2,3]],[[1,2,3]]]`

Example: Partitions

- Compute all partitions of a list `l`
- Partition of `l` : collection of nonempty lists `l1`, `l2`, ..., `lk` such that `l == l1 ++ l2 ++ ... ++ lk`

- `part [1,2,3] =`
  `[[[1],[2],[3]],[[1,2],[3]],[[1],[2,3]],[[1,2,3]]]`

- ```
  part :: [a] -> [[a]]
  part [x] = [[x]]
  part (x:xs) =
              [(x:head l):(tail l)  | l <- part xs]
            ++ [[x]:l | l <- part xs]
  ```
  ```

Summary

- Many cumulative operations on lists can be expressed in terms of folding a function through the list
- Built in functions `foldr, foldr1, foldl`
- Other higher order functions on lists:
  - `takeWhile`
  - `zipWith`
  - `zip`
- Several examples of functions on lists

Idiomatic programming

Idiomatic programming

- Programming languages are … languages!

Idiomatic programming

- Programming languages are … languages!
- Like “natural languages”, we can say the same thing in many ways

Idiomatic programming

- Programming languages are … languages!
- Like “natural languages”, we can say the same thing in many ways
- Initially, we use a language in its simplest and most direct form

Idiomatic programming

- Programming languages are … languages!
- Like “natural languages”, we can say the same thing in many ways
- Initially, we use a language in its simplest and most direct form
- As we master the language, we learn to use it idiomatically and more effectively

Idiomatic programming

- Programming languages are … languages!
- Like “natural languages”, we can say the same thing in many ways
- Initially, we use a language in its simplest and most direct form
- As we master the language, we learn to use it idiomatically and more effectively
- To learn a language, we must practice speaking it