Introduction to Programming: Lecture 4

K Narayan Kumar

Chennai Mathematical Institute

http://www.cmi.ac.in/~kumar

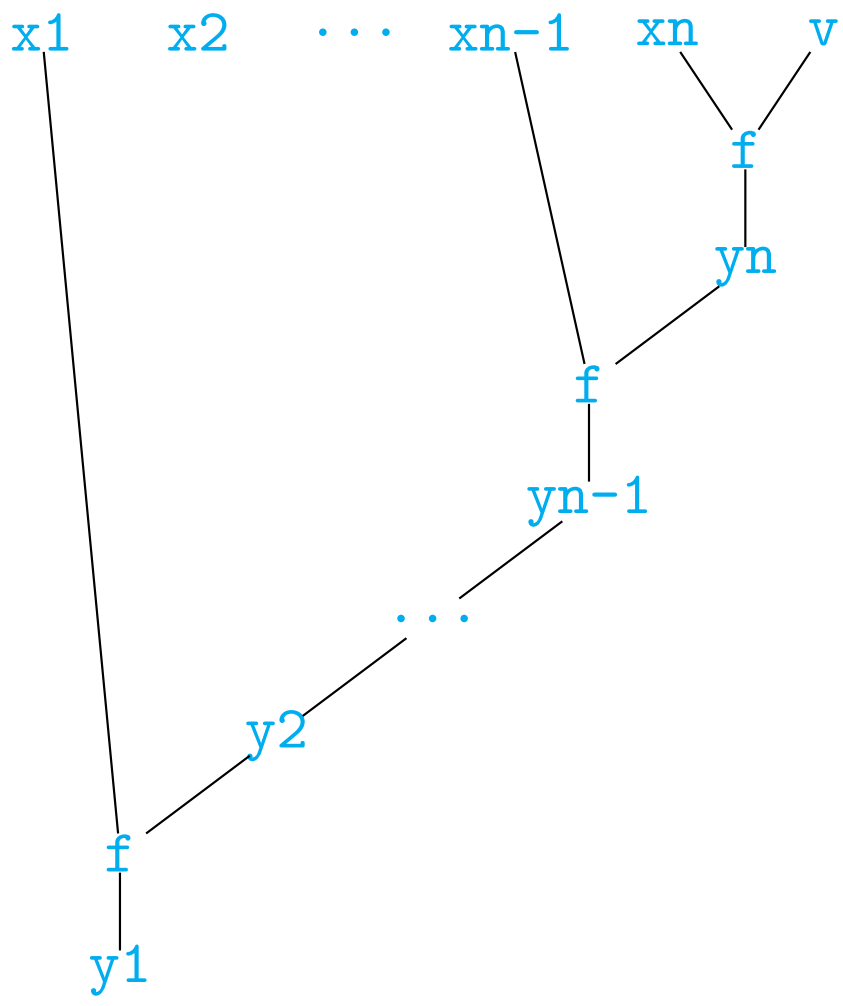
20 Aug 2013

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)
```



Combining the elements of List

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

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

Combining the elements of List

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

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

- Rewritten using `foldr`

```
sumlist ls  = foldr (+) 0 ls
multlist ls = foldr (*) 1 ls
```

mylength via foldr

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

mylength :: [Int] -> Int
mylength ls =  foldr f 0 ls

f x y = y+1
```

`foldr`: more examples

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

`foldr`: more examples

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

foldr appendright [] ls = ??
```

`foldr`: more examples

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

foldr appendright [] ls = ??

foldr appendright [] ls  = reverse  ls
```

`foldr`: more examples

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

foldr appendright [] ls = ??

foldr appendright [] ls  = reverse  ls
```

- What does the following calculate?

```
foldr (++) []
```

`foldr`: more examples

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

foldr appendright [] ls = ??

foldr appendright [] ls  = reverse  ls
```

- What does the following calculate?

  ```
  foldr (++) []
  ```

  Transforms a list of lists into a list by dissolving one level of brackets.

`foldr`: more examples

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

foldr appendright [] ls = ??

foldr appendright [] ls  = reverse  ls
```

- What does the following calculate?

  `foldr (++) []`

  Transforms a list of lists into a list by dissolving one level of brackets.

- The haskell function `concat`.

foldr

- What is the type of `foldr`?

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

foldr

- What is the type of `foldr`?

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

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

foldr

- What is the type of `foldr`?

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

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

Advantages of using Higher-order functions

- Allows reuse of code. Fewer bugs.

Advantages of using Higher-order functions

- Allows reuse of code. Fewer bugs.
- Makes code easier to read.

Advantages of using Higher-order functions

- Allows reuse of code. Fewer bugs.
- Makes code easier to read.
- Allows you to define your own programming language!

The function `foldr1`

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

The function `foldr1`

- Sometimes there is no natural value to assign to the empty list.
  - For instance in finding the maximum in list.
  - If the list is empty the answer should be undefined.

The function `foldr1`

- Sometimes there is no natural value to assign to the empty list.
  - For instance in finding the maximum in list.
  - If the list is empty the answer should be undefined.

- The Haskell function `foldr1` works very much like `foldr` but works only on nonempty lists.

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

The function `foldr1`

- Sometimes there is no natural value to assign to the empty list.
  - For instance in finding the maximum in list.
  - If the list is empty the answer should be undefined.

- The Haskell function `foldr1` works very much like `foldr` but works only on nonempty lists.

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

- `maxlist = foldr1 max`

Example: Position of a letter

- Given a letter `c` and a string `s` find the position of the left most occurnace of `c` in `s`.

Example: Position of a letter

- Given a letter `c` and a string `s` find the position of the left most occurnace of `c` in `s`.

```
position 'a' "battle axe" = 1
position 'd' "battle axe" = 10
```

Example: Position of a letter

- Given a letter `c` and a string `s` find the position of the left most occurnace of `c` in `s`.

```
position 'a' "battle axe" = 1
position 'd' "battle axe" = 10

position c []     = 0
position c (x:xs)
  | (x == c)    = 0
  | otherwise   = 1 + (position c xs)
```

`position` ...

- Position using an accumulator.

position ...

- Position using an accumulator.

```
posAux :: Int -> Char -> String -> Int
posAux i c [] = i
posAux i c (x:xs)
  | c == x      = i
  | otherwise   = posAux (i+1) c xs
```

position ...

- Position using an accumulator.

```
posAux :: Int -> Char -> String -> Int
posAux i c [] = i
posAux i c (x:xs)
  | c == x      = i
  | otherwise   = posAux (i+1) c xs

position c s = posAux 0 c s
```

Nesting Functions using `where`

- `position` using `where`

Nesting Functions using `where`

- `position` using `where`

```
position :: Char -> String -> Int
position c s = posAux 0 c s
  where
  posAux :: Int -> Char -> String -> Int
  posAux i c [] = i
  posAux i c (x:xs)
    | c == x      = i
    | otherwise   = posAux (i+1) c xs
```

Nesting Functions using `where`

- `position` using `where`

```
position :: Char -> String -> Int
position c s = posAux 0 c s
  where
  posAux :: Int -> Char -> String -> Int
  posAux i c [] = i
  posAux i c (x:xs)
    | c == x      = i
    | otherwise   = posAux (i+1) c xs
```

- `posAux` is not visible outside `position`.

Avoiding arguments via `where`

- Names in any enclosing block are visible.

Avoiding arguments via `where`

- Names in any enclosing block are visible.

```
position :: Char -> String -> Int
position c s = posAux 0 s
  where
  posAux :: Int -> String -> Int
  posAux i [] = i
  posAux i (x:xs)
    | c == x      = i
    | otherwise   = posAux (i+1) xs
```

Avoiding arguments via `where`

- Names in any enclosing block are visible.

```
position :: Char -> String -> Int
position c s = posAux 0 s
  where
  posAux :: Int -> String -> Int
  posAux i [] = i
  posAux i (x:xs)
    | c == x      = i
    | otherwise   = posAux (i+1) xs
```

- Reduces the number of arguments that need to be used.

The function `takeWhile`

```
takeWhile ::  (a -> Bool) -> [a] -> [a]
takeWhile (> 7) [8,1,9,10] = [8]
takeWhile (< 10) [8,1,9,10]= [8,1,9]
```

The function `takeWhile`

```
takeWhile ::  (a -> Bool) -> [a] -> [a]
takeWhile (> 7) [8,1,9,10] = [8]
takeWhile (< 10) [8,1,9,10]= [8,1,9]
```

- Write `position` using `takeWhile`

The function `takeWhile`

```
takeWhile ::  (a -> Bool) -> [a] -> [a]
takeWhile (> 7) [8,1,9,10] = [8]
takeWhile (< 10) [8,1,9,10]= [8,1,9]
```

- Write `position` using `takeWhile`

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

The function `takeWhile`

```
takeWhile ::  (a -> Bool) -> [a] -> [a]
takeWhile (> 7) [8,1,9,10] = [8]
takeWhile (< 10) [8,1,9,10]= [8,1,9]
```

- Write `position` using `takeWhile`

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

- There is also a corresponding function `dropWhile`

`zipWith` function

- Recall that `map` applies a function to every element of a list.

`zipWith` function

- Recall that `map` applies a function to every element of a list.
- `zipWith` works similarly on two lists.

`zipWith` function

- Recall that `map` applies a function to every element of a list.
- `zipWith` works similarly on two lists.

```
zipWith (+) [1,2,3] [7,8,6] = [8,10,9]
zipWith (+) [1,2,3] [7,8] = [8,10]
```

`zipWith` function

- Recall that `map` applies a function to every element of a list.
- `zipWith` works similarly on two lists.

```
zipWith (+) [1,2,3] [7,8,6] = [8,10,9]
zipWith (+) [1,2,3] [7,8] = [8,10]

zipWith (<) [1,2,3] [2,1,4] = [True,False,True]
```

`zipWith` function

- Recall that `map` applies a function to every element of a list.
- `zipWith` works similarly on two lists.

```
zipWith (+) [1,2,3] [7,8,6] = [8,10,9]
zipWith (+) [1,2,3] [7,8] = [8,10]

zipWith (<) [1,2,3] [2,1,4] = [True,False,True]

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

Example: Mark Lists

- `marks` – a list of list of integers
- Each list in `marks` corresponds to an assignment. The ith number is the mark obtained by the ith student.
- Compute the total marks obtained by each student.

Example: Mark Lists

- `marks` – a list of list of integers
- Each list in `marks` corresponds to an assignment. The *i*th number is the mark obtained by the *i*th student.
- Compute the total marks obtained by each student.

```
addMarks [[10,10,8], [9,2,10], [8,2,8], [3,7,8]]
= [30,21,34]
```

Example: Mark Lists

- `marks` – a list of list of integers
- Each list in `marks` corresponds to an assignment. The *i*th number is the mark obtained by the *i*th student.
- Compute the total marks obtained by each student.

```
addMarks [[10,10,8], [9,2,10], [8,2,8], [3,7,8]]
= [30,21,34]

addMarks [[3,4],[2]] = [5]
```

Example: Mark Lists

- `marks` – a list of list of integers
- Each list in `marks` corresponds to an assignment. The *i*th number is the mark obtained by the *i*th student.
- Compute the total marks obtained by each student.

```
addMarks [[10,10,8], [9,2,10], [8,2,8], [3,7,8]]
= [30,21,34]

addMarks [[3,4],[2]] = [5]
```

- Write down Haskell code for `addMarks`

Example: Mark Lists

- `marks` – a list of list of integers
- Each list in `marks` corresponds to an assignment. The *i*th number is the mark obtained by the *i*th student.
- Compute the total marks obtained by each student.

```
addMarks [[10,10,8], [9,2,10], [8,2,8], [3,7,8]]
= [30,21,34]

addMarks [[3,4],[2]] = [5]
```

- Write down Haskell code for `addMarks`

```
addMarks [x] = x
addMarks (x:xs) = zipWith (+) x (addMarks xs)
```

Example: Mark Lists

- `marks` – a list of list of integers
- Each list in `marks` corresponds to an assignment. The *i*th number is the mark obtained by the *i*th student.
- Compute the total marks obtained by each student.

```
addMarks [[10,10,8], [9,2,10], [8,2,8], [3,7,8]]
= [30,21,34]
```

```
addMarks [[3,4],[2]] = [5]
```

- Write down Haskell code for `addMarks`

```
addMarks [x] = x
addMarks (x:xs) = zipWith (+) x (addMarks xs)
```

or equivalently

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

Mark Lists ...

- `marks` – as before.
- Rearrange it so that you have a list of lists in which each list gives the marks obtained by one student.

```
transpose [[10,10,8], [9,2,10]]
      = [[10,9],[10,2],[8,10]]

transpose [[3,4],[2]] = undefined
```

Mark Lists ...

- `marks` – as before.
- Rearrange it so that you have a list of lists in which each list gives the marks obtained by one student.

```
transpose [[10,10,8], [9,2,10]]
      = [[10,9],[10,2],[8,10]]

transpose [[3,4],[2]] = undefined
```

- Write a haskell function to do this.