Introduction to Programming: Lecture 3

K Narayan Kumar

Chennai Mathematical Institute

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

13 Aug 2013

Polymorphism in Haskell

```
mylength [] = 0
mylength (x:xs) = 1 + mylength xs
```

Polymorphism in Haskell

```
mylength [] = 0
mylength (x:xs) = 1 + mylength xs

myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++ [x]
```

Polymorphism in Haskell

```
mylength [] = 0
mylength (x:xs) = 1 + mylength xs

myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++ [x]

myinit [x] = []
myinit (x:xs) = x:(myinit xs)
```

Polymorphism in Haskell

```
mylength [] = 0
mylength (x:xs) = 1 + mylength xs
```

```
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++ [x]
```

```
myinit [x] = []
myinit (x:xs) = x:(myinit xs)
```

- These functions work over any list.

Polymorphism in Haskell

```
mylength :: [a] -> Int
mylength [] = 0
mylength (x:xs) = 1 + mylength xs

myreverse :: [a] -> [a]
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++ [x]

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

The datatype `Char`

- Written within single quotes
  - `'a'`, `'3'`, `'%'`, `'#'`, ...

The datatype `Char`

- Written within single quotes
  - `'a'`, `'3'`, `'%'`, `'#'`, ...
- As usual, characters are stored as a table (e.g., ASCII)

The datatype `Char`

- Written within single quotes
  - `'a'`, `'3'`, `'%'`, `'#'`, ...
- As usual, characters are stored as a table (e.g., ASCII)
  - Functions `ord` and `chr` to go between characters and table

```
ord :: Char -> Int
chr :: Int -> Char
```

The datatype `Char`

- Written within single quotes
  - `'a'`, `'3'`, `'%'`, `'#'`, ...
- As usual, characters are stored as a table (e.g., ASCII)
  - Functions `ord` and `chr` to go between characters and table
    ```
    ord :: Char -> Int
    chr :: Int -> Char
    ```
  - These functions are inverses of each other
    `c == chr(ord c)` and `i == ord (chr i)`
    Note: Need to `import Char` to use these

The datatype `Char`

- Written within single quotes
  - `'a'`, `'3'`, `'%'`, `'#'`, ...
- As usual, characters are stored as a table (e.g., ASCII)
  - Functions `ord` and `chr` to go between characters and table
    ```
    ord :: Char -> Int
    chr :: Int -> Char
    ```
  - These functions are inverses of each other
    `c == chr(ord c)` and `i == ord (chr i)`
    Note: Need to `import Char` to use these
  - Assume `'a'`, `'b'`, ..., `'z'` occur consecutively.
  - Assume `'A'`, `'B'`, ..., `'Z'` occur consecutively.
  - Assume `'0'`, `'1'`, ..., `'9'` occur consecutively.

Functions using `Char`

- `capitalize` converts `'a'` to `'A'` etc

Functions using `Char`

- `capitalize` converts `'a'` to `'A'` etc
- A "brute force" solution using pattern matching

```
capitalize :: Char -> Char
capitalize 'a' = 'A'
capitalize 'b' = 'B'
...
capitalize 'z' = 'Z'
capitalize c = c
```

Functions using `Char`

- `capitalize` converts `'a'` to `'A'` etc
- A "brute force" solution using pattern matching

```
capitalize :: Char -> Char
capitalize 'a' = 'A'
capitalize 'b' = 'B'
...
capitalize 'z' = 'Z'
capitalize c = c
```

- A smarter solution: `'a'`,...,`'z'` and `'A'`,...,`'Z'` are contiguous

Functions using `Char`

- `capitalize` converts `'a'` to `'A'` etc
- A “brute force” solution using pattern matching

```
capitalize :: Char -> Char
capitalize 'a' = 'A'
capitalize 'b' = 'B'
...
capitalize 'z' = 'Z'
capitalize c = c
```

- A smarter solution: `'a'`,...,`'z'` and `'A'`,...,`'Z'` are contiguous

```
capitalize :: Char -> Char
capitalize c
  | ('a' <= c && c <= 'z') =
          chr (ord c + (ord 'A' - ord 'a'))
  | otherwise                = c
```

Strings

- `String` is a synonym for `[Char]`
  - Can write `['h','e','l','l','o']` as `"hello"`

Strings

- `String` is a synonym for `[Char]`
  - Can write `['h','e','l','l','o']` as `"hello"`
- All list functions work on `String`
  - `length`, `reverse`, `++`, ...

Strings

- `String` is a synonym for `[Char]`
  - Can write `['h','e','l','l','o']` as `"hello"`
- All list functions work on `String`
  - `length`, `reverse`, `++`, ...
- Check if a character exists in a string

```
exists :: Char -> String -> Bool
```

Strings

- `String` is a synonym for `[Char]`
  - Can write `['h','e','l','l','o']` as `"hello"`
- All list functions work on `String`
  - `length`, `reverse`, `++`, ...
- Check if a character exists in a string

```
exists :: Char -> String -> Bool

exists c "" = False
exists c (x:xs)
  | c == x     = True
  | otherwise = exists c xs
```

Strings

- `String` is a synonym for `[Char]`
  - Can write `['h','e','l','l','o']` as `"hello"`
- All list functions work on `String`
  - `length`, `reverse`, `++`, ...
- Check if a character exists in a string

```
exists :: Char -> String -> Bool

exists c "" = False
exists c (x:xs)
  | c == x    = True
  | otherwise = exists c xs
```

- Convert a string to uppercase

```
touppercase :: String -> String
```

Strings

- `String` is a synonym for `[Char]`
  - Can write `['h','e','l','l','o']` as `"hello"`
- All list functions work on `String`
  - `length`, `reverse`, `++`, ...
- Check if a character exists in a string

```
exists :: Char -> String -> Bool

exists c "" = False
exists c (x:xs)
  | c == x     = True
  | otherwise = exists c xs
```

- Convert a string to uppercase

```
touppercase :: String -> String

touppercase "" = ""
touppercase (c:cs) = (capitalize c):(touppercase cs)
```

List functions: `map`

- `touppercase` applies `capitalize` to each character in list

List functions: `map`

- `touppercase` applies `capitalize` to each character in list

```
sqrlist :: [Int] -> [Int]
sqrlist [] = []
sqrlist (x:xs) = sqr x : (sqrlist xs)
```

List functions: `map`

- `touppercase` applies `capitalize` to each character in list

```
sqrlist :: [Int] -> [Int]
sqrlist [] = []
sqrlist (x:xs) = sqr x : (sqrlist xs)
```

- `sqrlist` applies `sqr` to each number in the list

List functions: `map`

- `touppercase` applies `capitalize` to each character in list

```
sqrlist :: [Int] -> [Int]
sqrlist [] = []
sqrlist (x:xs) = sqr x : (sqrlist xs)
```

- `sqrlist` applies `sqr` to each number in the list
- Builtin function `map`

```
map f [x0,x1,..,xk] = [(f x0),(f x1),...,(f xk)]
```

List functions: `map`

- `touppercase` applies `capitalize` to each character in list

  ```
  sqrlist :: [Int] -> [Int]
  sqrlist [] = []
  sqrlist (x:xs) = sqr x : (sqrlist xs)
  ```

- `sqrlist` applies `sqr` to each number in the list
- Builtin function `map`

  ```
  map f [x0,x1,..,xk] = [(f x0),(f x1),...,(f xk)]
  ```

- Note that first argument of `map` is a function.

List functions: map

- touppercase applies capitalize to each character in list

```
sqrlist :: [Int] -> [Int]
sqrlist [] = []
sqrlist (x:xs) = sqr x : (sqrlist xs)
```

- sqrlist applies sqr to each number in the list
- Builtin function map

```
map f [x0,x1,..,xk] = [(f x0),(f x1),...,(f xk)]
```

- Note that first argument of map is a function.

  Higher-order functions.

Examples

- Recall that the type of `+` is `Int -> Int -> Int`

Examples

- Recall that the type of `+` is `Int -> Int -> Int`
- So, `+` applied to an integer `n` gives a function of type `Int -> Int`

Examples

- Recall that the type of `+` is `Int -> Int -> Int`
- So, `+` applied to an integer `n` gives a function of type

  `Int -> Int`
- The notation for this function is `(+ n)`

```
map (+ 3) [2,6,8] = [5,9,11]
map (* 2) [2,6,8] = [4,12,16]
```

Examples

- Sum of the length of the lists in the given list of lists.

```
sumLength::  [[a]] -> Int
```

Examples

- Sum of the length of the lists in the given list of lists.

```
sumLength::  [[a]] -> Int

sumLength [] = 0
sumLength (x:xs) = length x + (sumLength xs)
```

Examples

- Sum of the length of the lists in the given list of lists.

```
sumLength::  [[a]] -> Int

sumLength [] = 0
sumLength (x:xs) = length x + (sumLength xs)
```

- Can be written using `map` as:

```
sumLength l = sum (map length l)
```

The function `map`

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

The function `map`

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

- What is the type of `map`?

The function `map`

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

- What is the type of `map`?

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

List functions: `filter`

- Select items from a list based on a property
- `filter` selects all items from `l` that satisfy `p`

```
filter p [] = []
filter p (x:xs)
  | (p x)      = x:(filter p xs)
  | otherwise = filter p xs
```

List functions: `filter`

- Select items from a list based on a property
- `filter` selects all items from `l` that satisfy `p`

```
filter p [] = []
filter p (x:xs)
  | (p x)     = x:(filter p xs)
  | otherwise = filter p xs
```

- `filter` is also an higher-order function.

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

List functions: `filter`

- Select items from a list based on a property
- `filter` selects all items from `l` that satisfy `p`

```
filter p [] = []
filter p (x:xs)
  | (p x)      = x:(filter p xs)
  | otherwise = filter p xs
```

- `filter` is also an higher-order function.

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

- `evenonly l = filter iseven l`

```
iseven :: Int -> Bool
iseven n = (mod n 2 == 0)
```

List functions: `filter`

- Select items from a list based on a property
- `filter` selects all items from `l` that satisfy `p`

```
filter p [] = []
filter p (x:xs)
  | (p x)      = x:(filter p xs)
  | otherwise = filter p xs
```

- `filter` is also an higher-order function.

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

- `evenonly l = filter iseven l`

```
iseven :: Int -> Bool
iseven n = (mod n 2 == 0)
```

Combining `map` and `filter`

- Extract all the vowels in the input and captialize them.

Combining `map` and `filter`

- Extract all the vowels in the input and captialize them.
- Solution: Use filter to extract the vowels and use map to capitalize them.

Combining `map` and `filter`

- Extract all the vowels in the input and captialize them.
- Solution: Use filter to extract the vowels and use map to capitalize them.

```
capvow :: [Char] -> [Char]
capvow l = map touppercase (filter isvowel l)

isvowel :: Char -> Char
isvowel c =  (c=='a') || (c=='e') || (c=='i')
                || (c=='o') || (c=='u')
```

Combining `map` and `filter`

- The list of squares of even numbers in `ls`

Combining `map` and `filter`

- The list of squares of even numbers in `ls`

```
sqreven :: [Int] -> [Int]
sqreven  ls = map sqr (filter iseven ls)
```

Example: Counting words

- A function `wordc` to count the number of words in a given string.

Example: Counting words

- A function `wordc` to count the number of words in a given string.
- Assume that words are separated by white spaces. i.e. `' '`, `'\t'`, `'\n'`

Example: Counting words

- A function `wordc` to count the number of words in a given string.
- Assume that words are separated by white spaces. i.e.
  `' ', '\t', '\n'`
- Suppose

```
whitespace ' '  = True
whitespace '\t' = True
whitespace '\n' = True
whitespace _    = False
```

Example: Counting words

- A function `wordc` to count the number of words in a given string.
- Assume that words are separated by white spaces. i.e.
  `' ', '\t', '\n'`
- Suppose

```
whitespace ' '  = True
whitespace '\t' = True
whitespace '\n' = True
whitespace _    = False
```

- Can we simply count the number of white spaces?

Example: Counting words

- A function `wordc` to count the number of words in a given string.
- Assume that words are separated by white spaces. i.e.
  `' ', '\t', '\n'`
- Suppose

  ```
  whitespace ' '  = True
  whitespace '\t' = True
  whitespace '\n' = True
  whitespace _    = False
  ```
- Can we simply count the number of white spaces?
- NO. Consider `"abc      d"`.

Word Count: counting whitespace

- `wspace` counts the number of whitespaces in the given input.
- ```
  wspace [] = 0
  wspace (x:xs)
     | whitespace x = 1 + wspace xs
     | otherwise    = wspace xs
  ```
  ```

Word Count: counting whitespace

- `wspace` counts the number of whitespaces in the given input.
- ```
  wspace [] = 0
  wspace (x:xs)
    | whitespace x = 1 + wspace xs
    | otherwise    = wspace xs
  ```
- `wspace l = length (filter whitespace l)`
  ```

Word Count

- If you are outside (any word) then whitespace characters can be ignored.

Word Count

- If you are outside (any word) then whitespace characters can be ignored.
- If you are inside a word then non-whitespace characters can be ignored.

Word Count

- If you are outside (any word) then whitespace characters can be ignored.
- If you are inside a word then non-whitespace characters can be ignored.
- If you are outside and you encounter a non-whitespace it marks the beginning of a word.

Word Count

- If you are outside (any word) then whitespace characters can be ignored.
- If you are inside a word then non-whitespace characters can be ignored.
- If you are outside and you encounter a non-whitespace it marks the beginning of a word.
- If you are inside and you encouter a whitespace it marks the ending of a word.

Word Count

- If you are outside (any word) then whitespace characters can be ignored.
- If you are inside a word then non-whitespace characters can be ignored.
- If you are outside and you encounter a non-whitespace it marks the beginning of a word.
- If you are inside and you encouter a whitespace it marks the ending of a word.
- Count the number of word beginnings.

Word Count: general case ...

- Functions to keep track of the current position (inside or outside a word.)

Word Count: general case ...

- Functions to keep track of the current position (inside or outside a word.)

```
inwordAux :: Int -> String -> Int
inwordAux i [] = i
inwordAux i (c:cs)
  | whitespace c   = outwordAux i cs
  | otherwise       = inwordAux i cs

outwordAux :: Int -> String -> Int
outwordAux i [] = i
outwordAux i (c:cs)
  | whitespace c = outwordAux i cs
  | otherwise     = inwordAux (i+1) cs
```

Word Count: general case ...

- Functions to keep track of the current position (inside or outside a word.)

```
inwordAux :: Int -> String -> Int
inwordAux i [] = i
inwordAux i (c:cs)
  | whitespace c  = outwordAux i cs
  | otherwise      = inwordAux i cs

outwordAux :: Int -> String -> Int
outwordAux i [] = i
outwordAux i (c:cs)
  | whitespace c = outwordAux i cs
  | otherwise     = inwordAux (i+1) cs

wordc l = outwordAux 0 l
```

Word Count: A direct solution

- The action to be taken on reading a character also depends on the previous character.

Word Count: A direct solution

- The action to be taken on reading a character also depends on the previous character.
- Suppose `w` denotes whitespace and `c` denotes a non-white space, then

Word Count: A direct solution

- The action to be taken on reading a character also depends on the previous character.
- Suppose `w` denotes whitespace and `c` denotes a non-white space, then

Previous	Current	Action
`w`	`w`	Do Nothing
`w`	`c`	Increment
`c`	`c`	Do Nothing
`c`	`w`	Do Nothing

Word Count: A direct solution

- The action to be taken on reading a character also depends on the previous character.
- Suppose `w` denotes whitespace and `c` denotes a non-white space, then

Previous	Current	Action
`w`	`w`	Do Nothing
`w`	`c`	Increment
`c`	`c`	Do Nothing
`c`	`w`	Do Nothing

- An Haskell implementation:

```
wordcAux (x:y:xs)
  | ws x && not (ws y) = 1 + wordcAux (y:xs)
  | otherwise           = wordcAux (y:xs)
```

Word Count: A direct solution

- The action to be taken on reading a character also depends on the previous character.
- Suppose `w` denotes whitespace and `c` denotes a non-white space, then

Previous	Current	Action
`w`	`w`	Do Nothing
`w`	`c`	Increment
`c`	`c`	Do Nothing
`c`	`w`	Do Nothing

- An Haskell implementation:

```
wordcAux [x] = 0
wordcAux (x:y:xs)
   | ws x && not (ws y) = 1 + wordcAux (y:xs)
   | otherwise          = wordcAux (y:xs)

wordc l = wordcAux (' ':l)
```

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)
```

- What is the common pattern across these definitions?

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)
```

- What is the common pattern across these definitions?

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

- What is the common pattern across these definitions?

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

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

`foldr` is the Library version of `combine`

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

`foldr` is the Library version of `combine`

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