Introduction to Programming: Lecture 2

K Narayan Kumar

Chennai Mathematical Institute

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

08 August 2013

Functions with multiple inputs ...

- Consider a function with many arguments

```
f x1 x2 ...xn = y
```

Functions with multiple inputs . . .

- Consider a function with many arguments

```
f x1 x2 ...xn = y
```

  - Suppose each input `xi` is of type `Int`
  - Suppose Output `y` is of type `Bool`
  - Type of `f` is

Functions with multiple inputs . . .

- Consider a function with many arguments

```
f x1 x2 ...xn = y
```

  - Suppose each input `xi` is of type `Int`
  - Suppose Output `y` is of type `Bool`
  - Type of `f` is

```
f::Int -> (Int -> ( ...(Int -> Bool) ...))
```

Functions with multiple inputs ...

- Consider a function with many arguments

  `f x1 x2 ...xn = y`

  - Suppose each input `xi` is of type `Int`
  - Suppose Output `y` is of type `Bool`
  - Type of `f` is

    `f::Int -> (Int -> ( ...(Int -> Bool) ...))`

- For convenience, we are allowed to write
  - `f x1 x2 ...xn`
    to mean
    `(...((f x1) x2) ...xn)`

Functions with multiple inputs . . .

- Consider a function with many arguments

  `f x1 x2 ...xn = y`

  - Suppose each input `xi` is of type `Int`
  - Suppose Output `y` is of type `Bool`
  - Type of `f` is

    `f::Int -> (Int -> ( ...(Int -> Bool) ...))`

- For convenience, we are allowed to write
  - `f x1 x2 ...xn`
    to mean
    `(...((f x1) x2) ...xn)`
  - `f ::  Int -> Int -> ...Int -> Bool`
    to mean
    `f ::  Int -> (Int -> ( ...(Int -> Bool) ...))`

Functions with multiple inputs ...

- Consider a function with many arguments

  `f x1 x2 ...xn = y`

  - Suppose each input `xi` is of type `Int`
  - Suppose Output `y` is of type `Bool`
  - Type of `f` is

    `f::Int -> (Int -> ( ...(Int -> Bool) ...))`

- For convenience, we are allowed to write
  - `f x1 x2 ...xn`

    to mean

    `(...((f x1) x2) ...xn)`
  - `f ::  Int -> Int -> ...Int -> Bool`

    to mean

    `f ::  Int -> (Int -> ( ...(Int -> Bool) ...))`
  - This works for any combination of input and output types

Functions in Haskell

- Pattern Matching

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * (factorial (n-1))
```

Functions in Haskell

- Pattern Matching

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * (factorial (n-1))
```

- Conditional definitions

```
factorial :: Int -> Int
factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 0 = n * (factorial (n-1))
```

Functions in Haskell

- Pattern Matching

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * (factorial (n-1))
```

- Conditional definitions

```
factorial :: Int -> Int
factorial 0 = 1
factorial n
   | n < 0 = factorial (-n)
   | n > 0 = n * (factorial (n-1))
```

- Using `otherwise`

```
xor :: Bool -> Bool -> Bool
xor b1 b2
   | b1 && not(b2) = True
   | not(b1) && b2 = True
   | otherwise     = False
```

Functions in Haskell

- Wild Cards.

```
or :: Bool -> Bool -> Bool
or True _   = True
or _   True = True
or _   _    = False
```

Functions in Haskell

- Wild Cards.

```
or :: Bool -> Bool -> Bool
or True _   = True
or _   True = True
or _   _    = False
```

- `_` matches anything, but cannot be used in the righthand side.

Functions in Haskell

- Wild Cards.

```
or :: Bool -> Bool -> Bool
or True _  = True
or _  True = True
or _  _    = False
```

- _ matches anything, but cannot be used in the righthand side.

```
or :: Bool -> Bool -> Bool
or False x =  x
or x  False = x
or _  _    =  True
```

Computation as rewriting

- Use definitions to simplify expressions till no further simplification is possible

Computation as rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications
  - `3 + 5` $\rightsquigarrow$ `8`
  - `True || False` $\rightsquigarrow$ `True`

Computation as rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications
  - `3 + 5` $\rightsquigarrow$ `8`
  - `True || False` $\rightsquigarrow$ `True`
- Simplifications based on user defined functions

Computation as rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications
  - `3 + 5` $\rightsquigarrow$ `8`
  - `True || False` $\rightsquigarrow$ `True`
- Simplifications based on user defined functions

```
power :: Int -> Int -> Int
power x 0 = 1
power x n = x * (power x (n-1))
```

Computation as rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications
  - `3 + 5` $\rightsquigarrow$ `8`
  - `True || False` $\rightsquigarrow$ `True`
- Simplifications based on user defined functions

```
power :: Int -> Int -> Int
power x 0 = 1
power x n = x * (power x (n-1))
```

- `power 3 2`
  $\rightsquigarrow$ `3 * (power 3 (2-1))`
  $\rightsquigarrow$ `3 * (power 3 1)`
  $\rightsquigarrow$ `3 * (3 * (power 3 (1-1)))`
  $\rightsquigarrow$ `3 * (3 * (power 3 0))`
  $\rightsquigarrow$ `3 * (3 * 1)`
  $\rightsquigarrow$ `3 * 3` $\rightsquigarrow$ `9`

Examples

- A function to calculate the gcd of two given numbers:

Examples

- A function to calculate the gcd of two given numbers:

```
mygcd:: Int -> Int -> Int
mygcd x 0 = x
mygcd x n
  | (x <= n) = mygcd x (n-x)
  | otherwise = mygcd n x
```

Largest Divisor

- A function to determine the largest divisor (other than itself) of a given number.

Largest Divisor

- A function to determine the largest divisor (other than itself) of a given number.

```
largediv :: Int -> Int
largediv n = divaux n (n-1)

divaux :: Int -> Int -> Int
divaux i j
   | (mod i j  == 0)    =  j
   | otherwise           = divaux i (j-1)
```

Example: Approximating the logarithm

- $\log_k n$ is the number of times we can divide n by k before we reach 1

Example: Approximating the logarithm

- $\log_k n$ is the number of times we can divide n by k before we reach 1
- Integer approximation: number of times we can divide n by k without going strictly below 1

Example: Approximating the logarithm

- $\log_k n$ is the number of times we can divide n by k before we reach 1
- Integer approximation: number of times we can divide n by k without going strictly below 1
- $\log_2 30 \approx 4$ because $\frac{30}{2^4} > 1$ but $\frac{30}{2^5} < 1$

Example: Approximating the logarithm

- $\log_k n$ is the number of times we can divide n by k before we reach 1
- Integer approximation: number of times we can divide n by k without going strictly below 1
- $\log_2 30 \approx 4$ because $\frac{30}{2^4} > 1$ but $\frac{30}{2^5} < 1$
- Keep dividing n by k till we reach 1

Example: Approximating the logarithm

- $\log_k n$ is the number of times we can divide n by k before we reach 1
- Integer approximation: number of times we can divide n by k without going strictly below 1
- $\log_2 30 \approx 4$ because $\frac{30}{2^4} > 1$ but $\frac{30}{2^5} < 1$
- Keep dividing n by k till we reach 1

```
mylog :: Int -> Int -> Int
mylog k 1 = 0
mylog k n = 1 + (mylog k (div n k))
```

Example: Approximating the logarithm

- $\log_k n$ is the number of times we can divide n by k before we reach 1
- Integer approximation: number of times we can divide n by k without going strictly below 1
- $\log_2 30 \approx 4$ because $\frac{30}{2^4} > 1$ but $\frac{30}{2^5} < 1$
- Keep dividing n by k till we reach 1

```
mylog :: Int -> Int -> Int
mylog k 1 = 0
mylog k n = 1 + (mylog k (div n k))
```

Oops!

Example: Approximating the logarithm

- $\log_k n$ is the number of times we can divide n by k before we reach 1
- Integer approximation: number of times we can divide n by k without going strictly below 1
- $\log_2 30 \approx 4$ because $\frac{30}{2^4} > 1$ but $\frac{30}{2^5} < 1$
- Keep dividing n by k till we reach 1, or go below 1!

```
mylog :: Int -> Int -> Int
mylog k 1 = 0
mylog k n
  | n >= k      = 1 + (mylog k (div n k))
  | otherwise   = 0
```

Example: Reversing the digits in an integer

- `intreverse 13276` $\rightsquigarrow$ `67231`

Example: Reversing the digits in an integer

- `intreverse 13276` $\rightsquigarrow$ `67231`
- Strategy
  - Split `13276` as `1327` and `6` using `div 13276 10` and `mod 13276 10`

Example: Reversing the digits in an integer

- `intreverse 13276` $\rightsquigarrow$ `67231`
- Strategy
  - Split `13276` as `1327` and `6` using `div 13276 10` and `mod 13276 10`
  - Recursively reverse `1327`

Example: Reversing the digits in an integer

- `intreverse 13276` $\rightsquigarrow$ `67231`
- Strategy
  - Split `13276` as `1327` and `6` using `div 13276 10` and `mod 13276 10`
  - Recursively reverse `1327`
  - Multiply `6` by appropriate power of `10` and add

Example: Reversing the digits in an integer

- `intreverse 13276` $\rightsquigarrow$ `67231`
- Strategy
  - Split `13276` as `1327` and `6` using `div 13276 10` and `mod 13276 10`
  - Recursively reverse `1327`
  - Multiply `6` by appropriate power of `10` and add
    - Use `mylog` to decide the power of `10` to use

Example: Reversing the digits in an integer

- `intreverse 13276` $\rightsquigarrow$ `67231`
- Strategy
  - Split `13276` as `1327` and `6` using `div 13276 10` and `mod 13276 10`
  - Recursively reverse `1327`
  - Multiply `6` by appropriate power of `10` and add
    - Use `mylog` to decide the power of `10` to use

```
intreverse :: Int -> Int
intreverse n
  | n < 10      = n
  | otherwise   = (intreverse (div n 10)) +
                  (mod n 10)*(power 10 (mylog 10 n))
```

Lists

- To describe a collection of values in Haskell, use a list
  - `[1,2,3,1]` is a list of `Int`
  - `[True,False,True]` is a list of `Bool`

Lists

- To describe a collection of values in Haskell, use a list
  - `[1,2,3,1]` is a list of `Int`
  - `[True,False,True]` is a list of `Bool`
- Elements of a list must all be of one type
  - Cannot write `[1,2,True]` or `[3,'a']`

Lists

- To describe a collection of values in Haskell, use a list
  - `[1,2,3,1]` is a list of `Int`
  - `[True,False,True]` is a list of `Bool`
- Elements of a list must all be of one type
  - Cannot write `[1,2,True]` or `[3,'a']`
- List of underlying type `T` has type `[T]`
  - `[1,2,3,1]::[Int]`
  - `[True,False,True]::[Bool]`
- Empty list is `[]` for all types

Lists

- To describe a collection of values in Haskell, use a list
  - `[1,2,3,1]` is a list of `Int`
  - `[True,False,True]` is a list of `Bool`
- Elements of a list must all be of one type
  - Cannot write `[1,2,True]` or `[3,'a']`
- List of underlying type `T` has type `[T]`
  - `[1,2,3,1]::[Int]`
  - `[True,False,True]::[Bool]`
- Empty list is `[]` for all types
- Lists can be nested
  - `[[3,2],[],[7,7,7]]` is of type `[[Int]]`

Internal representation on lists

- Basic list building operator is `:`
  - Append an element to the left of a list
  - `1:[2,3,4]` $\rightsquigarrow$ `[1,2,3,4]`

Internal representation on lists

- Basic list building operator is `:`
  - Append an element to the left of a list
  - `1:[2,3,4]` $\rightsquigarrow$ `[1,2,3,4]`
- All Haskell lists are built up from `[]` using operator `:`
  - `[1,2,3,4]` is actually `1:(2:(3:(4:[])))`
  - `:` is right associative, so `1:2:3:4:[]` = `1:(2:(3:(4:[])))`

Internal representation on lists

- Basic list building operator is `:`
  - Append an element to the left of a list
  - `1:[2,3,4]` $\rightsquigarrow$ `[1,2,3,4]`
- All Haskell lists are built up from `[]` using operator `:`
  - `[1,2,3,4]` is actually `1:(2:(3:(4:[])))`
  - `:` is right associative, so `1:2:3:4:[]` = `1:(2:(3:(4:[])))`
- Functions `head` and `tail` to decompose a list
  - `head (x:l) = x`
  - `tail (x:l) = l`
  - Undefined for `[]`
  - `head` returns a value, `tail` returns a list

Defining list functions inductively

- Natural numbers are built up from `0` using `succ n`
- Define `f 0` explicitly and give a rule to compute `f (succ n)` by combining `n` and `f n`

Defining list functions inductively

- Natural numbers are built up from `0` using `succ n`
- Define `f 0` explicitly and give a rule to compute `f (succ n)` by combining `n` and `f n`
- Lists are built up from `[]` using `:`
- Define `f` for `[]`
- Compute `f l` by combining `head l` and `f (tail l)`

Defining list functions inductively

- Natural numbers are built up from `0` using `succ n`
- Define `f 0` explicitly and give a rule to compute `f (succ n)` by combining `n` and `f n`

- Lists are built up from `[]` using `:`
- Define `f` for `[]`
- Compute `f l` by combining `head l` and `f (tail l)`

```
mylength :: [Int] -> Int
```

Defining list functions inductively

- Natural numbers are built up from `0` using `succ n`
- Define `f 0` explicitly and give a rule to compute `f (succ n)` by combining `n` and `f n`
- Lists are built up from `[]` using `:`
- Define `f` for `[]`
- Compute `f l` by combining `head l` and `f (tail l)`

```
mylength :: [Int] -> Int

mylength [] = 0
mylength l  = 1 + (mylength (tail l))
```

Defining list functions inductively

- Natural numbers are built up from `0` using `succ n`
- Define `f 0` explicitly and give a rule to compute `f (succ n)` by combining `n` and `f n`

- Lists are built up from `[]` using `:`
- Define `f` for `[]`
- Compute `f l` by combining `head l` and `f (tail l)`

```
mylength :: [Int] -> Int

mylength [] = 0
mylength l  = 1 + (mylength (tail l))


mysum :: [Int] -> Int
```

Defining list functions inductively

- Natural numbers are built up from `0` using `succ n`
- Define `f 0` explicitly and give a rule to compute `f (succ n)` by combining `n` and `f n`

- Lists are built up from `[]` using `:`
- Define `f` for `[]`
- Compute `f l` by combining `head l` and `f (tail l)`

```
mylength :: [Int] -> Int

mylength [] = 0
mylength l  = 1 + (mylength (tail l))

mysum :: [Int] -> Int

mysum [] = 0
mysum l  = (head l) + (mysum (tail l))
```

Functions on lists ...

- Implicitly extract head and tail using pattern matching

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


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

Functions on lists . . .

- Append to the right: `appendright 1 [2,3]` $\leadsto$ `[2,3,1]`

Functions on lists . . .

- Append to the right: `appendright 1 [2,3]` $\leadsto$ `[2,3,1]`

```
appendright :: Int -> [Int] -> [Int]
appendright x [] = [x]
appendright x (y:ys) = y:(appendright x ys)
```

Functions on lists ...

- Append to the right: `appendright 1 [2,3]` $\rightsquigarrow$ `[2,3,1]`

  ```
  appendright :: Int -> [Int] -> [Int]
  appendright x [] = [x]
  appendright x (y:ys) = y:(appendright x ys)
  ```

- Combine two lists into one — `append`
  - `append [3,2] [4,6,7]` $\rightsquigarrow$ `[3,2,4,6,7]`

Functions on lists ...

- Append to the right: `appendright 1 [2,3]` $\leadsto$ `[2,3,1]`

```
appendright :: Int -> [Int] -> [Int]
appendright x [] = [x]
appendright x (y:ys) = y:(appendright x ys)
```

- Combine two lists into one — `append`
  - `append [3,2] [4,6,7]` $\leadsto$ `[3,2,4,6,7]`

```
append :: [Int] -> [Int] -> [Int]
append [] ys = ys
append (x:xs) ys = x:(append xs ys)
```

Functions on lists . . .

- Append to the right: `appendright 1 [2,3]` ⇝ `[2,3,1]`

```
appendright :: Int -> [Int] -> [Int]
appendright x [] = [x]
appendright x (y:ys) = y:(appendright x ys)
```

- Combine two lists into one — `append`
  - `append [3,2] [4,6,7]` ⇝ `[3,2,4,6,7]`

```
append :: [Int] -> [Int] -> [Int]
append [] ys = ys
append (x:xs) ys = x:(append xs ys)
```

- Builtin operator `++` for `append`
  - `[1,2,3] ++ [4,3]` ⇝ `[1,2,3,4,3]`

Functions on lists . . .

- Reversing a list

Functions on lists . . .

- Reversing a list

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

Functions on lists ...

- Reversing a list

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

- Check if a list of integers is sorted.

Functions on lists ...

- Reversing a list

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

- Check if a list of integers is sorted.

```
ascending :: [Int] -> Bool
ascending []  = True
ascending [x] = True
ascending (x:y:ys)
          | (x <= y)  = ascending (y:ys)
          | otherwise = False
```

Functions on Lists ...

- Check if a list of integers is alternating.

Functions on Lists ...

- Check if a list of integers is alternating.

```
alternating :: [Int] -> Bool
alternating l = (updown l) || (downup l)

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

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

Some built in functions on lists

- `head`, `tail`, `length`, `sum`, `reverse`, ...

Some built in functions on lists

- `head`, `tail`, `length`, `sum`, `reverse`, ...
- `init l` returns all but the last element of `l`
  `init [1,2,3]` $\rightsquigarrow$ `[1,2]`
  `init [2]` $\rightsquigarrow$ `[]`
- `last l` returns the last element in `l`
  `last [1,2,3]` $\rightsquigarrow$ `3`
  `last [2]` $\rightsquigarrow$ `2`

Some built in functions on lists

- `head`, `tail`, `length`, `sum`, `reverse`, ...
- `init l` returns all but the last element of `l`
  `init [1,2,3]` $\leadsto$ `[1,2]`
  `init [2]` $\leadsto$ `[]`
- `last l` returns the last element in `l`
  `last [1,2,3]` $\leadsto$ `3`
  `last [2]` $\leadsto$ `2`
- `take n l` returns first `n` values in `l`
- `drop n l` leaves out first `n` values in `l`

```
l == (take n l) ++ (drop n l)
```

Polymorphism

Consider the functions `length, reverse, init, ...`

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

Polymorphism

Consider the functions `length`, `reverse`, `init`, `...`

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

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

Polymorphism

Consider the functions `length, reverse, init, ...`

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

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

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

Polymorphism

Consider the functions `length`, `reverse`, `init`, `...`

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

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

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

None of these functions look into the elements of the list.

Polymorphism

Consider the functions `length, reverse, init, ...`

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

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

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

None of these functions look into the elements of the list.

In Haskell, these functions will work over lists of any type!

Polymorphism

Consider the functions `length, reverse, init, ...`

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

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

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

None of these functions look into the elements of the list.

In Haskell, these functions will work over lists of any type!

Polymorphic Functions

Polymorphism

Consider the functions `length, reverse, init, ...`

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

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

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

None of these functions look into the elements of the list.

In Haskell, these functions will work over lists of any type!

Polymorphic Functions

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